Personalisierbares dynamisches Diagrammauswahlmodul

Entwicklung einer personalisierbaren dynamischen Diagramm-Auswahl für das Extensible Workflow Management for Simulationes (EWMS)

Praxisbericht
Hendrik Jüchter

IB-Nummer 112-2013/41
Zugänglichkeitsstufe A/I

Institut für Flugführung
Direktor: Prof. Dr. Dirk Kügler

Dokument Information

Zuständiger Projekt- / Abteilungsleiter:	Dipl.-Ing. Alexander Scharnweber / Dipl.-Ing. Frank Knabe
Zuständiger Autor:	Hendrik Jüchter
Weitere Autor(en):	
Projekt / Zielfeld:	Entwicklung einer personalisierbaren dynamischen Diagramm-Auswahl für das Extensible Workflow Management for Simulations (EWMS)
Zugänglichkeitsstufe:	A / I
Datei:	IB-112-2013-41_Diagrammauswahl_001
Version:	0.01
Speicherdatum:	2013-09-04
Gesamtseitenzahl:	52

Änderungsverfolgung

Version	Datum	Geänderte Seiten / Kapitel	Bemerkungen
0.01	28.08.2013	Gesamtes Dokument	Formatierung der Praxisarbeit nach Institutsvorlage

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt und keine weiteren als die angegebenen Quellen und Hilfsmittel benutzt habe.

Braunschweig, den

_____

Unterschrift

Zusammenfassung

Am Institut für Flugführung des Deutschen Zentrums für Luft- und Raumfahrt in der Abteilung Luftverkehrssysteme werden Verkehrsabläufe im Nahbereich von Flughäfen mit dem Einsatz von Schnellzeitsimulationssystemen erforscht. Für die einheitliche Auswertung verschiedener Luftverkehrssimulationen wurde die Auswertungssoftware Extensible Workflow Management for Simulations (EWMS) entwickelt. Um die Leistungen und Möglichkeiten der Auswertungen zu verbessern, wird eine neue Version des EWMS entwickelt. Um den Aufwand der Entwicklung neuer Auswertungen zu verringern, wird im Rahmen dieser Praxisarbeit ein personalisierbares dynamisches Diagrammauswahlmodul für die zweite Generation des EWMS entwickelt.

Durch die Entwicklung eines Diagrammauswahlmoduls für alle Auswertungen muss nicht mehr für jede Auswertung eine individuelle Diagrammausgabe programmiert werden. Das Modul erstellt dem Anwender Diagramme sinnvoller Diagrammtypen auf Basis der einheitlichen Ausgangsdaten der Auswertungen. Der Anwender hat die Möglichkeit zwischen den Diagrammtypen auszuwählen und auch mehrere Diagramme gleichzeitig zu erzeugen und zu vergleichen. Außerdem gibt es eine automatische Diagrammauswahl, die den favorisierten Diagrammtyp des Nutzers direkt aufruft. Durch das Diagrammauswahlmodul können zukünftige Auswertungsmodule für das EWMS 2 einfacher und schneller entwickelt werden.

Abstract

The department of Air Transportation in the Institute of Flight Guidance of the German Aerospace Center investigates the traffic flows in the vicinity of airports using fast time simulation systems. For the uniform analysis of different air traffic management simulations the analysis software Extensible Workflow Management for Simulations (EWMS) was developed. In order to improve the performance and capabilities of the analysis, a new version of the EWMS is under development. To reduce the costs for development of new reports, a customizable dynamic diagram selection module is developed within this thesis and implemented in the second generation of the EWMS.

An individual graph output no longer needs to be implemented for every report any more thanks to the development of a generic diagram selection module for all reports. The module provides diagrams of meaningful diagram types based on the uniform output data of the reports. The user can select between the different diagram types. In addition, he has the opportunity to generate several diagrams at the same time and compare them. Moreover, there is an automatic diagram selection which directly displays the favored diagram type of the user. Using the diagram selection module, future report modules can be developed for the EWMS 2 easier and faster.

Inhaltsverzeichnis

Vorwort

Während meiner Praxisphasen arbeite ich im Deutschen Zentrum für Luft- und Raumfahrt e.V.DLR im Institut für Flugführung (FL) in der Abteilung Luftverkehrssysteme (LVS). Die Abteilung LVS ist zuständig für den Einsatz und die Weiterentwicklung von Schnellzeitsimulationen für Verkehrsabläufe und arbeitet an neuen Methoden zu Systemmodellierung und -analyse.[1]

Mein Betreuer arbeitet an dem Projekt EWMS - einer Softwareumgebung zur einheitlichen Auswertung verschiedener Schnellzeit- und Echtzeit-Luftverkehrssimulationen. Das Programm stellt dafür standardisierte Reports und Visualisierungen bereit. Die Abkürzung EWMS steht inoffiziell auch für „Eierlegende Wollmilchsau", denn das Programm soll, soweit möglich, alle Schritte des Auswertungs-Workflow abdecken.

Die Aufgaben, die ich in der dritten und vierten Praxisphase bearbeitet habe, werden im folgenden Bericht dargestellt.

[1] Beschreibung der Abteilung, vgl. `http://www.dlr.de/fl/desktopdefault.aspx/tabid-1136/`

Kapitel 1

Einleitung

Für die beiden Praxismodule drei und vier wird ein zukünftiger Bestandteil des EWMS erarbeitet. Für eine neue Version des EWMS wird ein Diagrammmodul geschrieben.

1.1 Aufgabenstellung

Im Rahmen dieser Praxisarbeit soll ein Diagramm-Modul für die zweite Generation des EWMS entwickelt werden. Das Modul soll existierende Schnittstellen des EWMS2 verwenden. Geeignete Diagramme sollen basierend auf den Eingangsdaten dynamisch ausgewählt werden. Die Eingangsdaten sollen in geeignete JFreeChart-Datenstrukturen transformiert werden. JFreeChart ist ein Werkzeug um Diagramme in Java zu erzeugen. Es sollen je nach Eingangsdaten ein oder mehrere Diagramme erzeugt werden. Des Weiteren soll zur Steuerung der Diagrammvorschläge das Nutzerverhalten verfolgt und gespeichert werden und eine grafische Oberfläche für die Diagrammauswahl entwickelt werden.

1.2 Motivation

Das Institut für Flugführung des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR) ist eine der weltweit führenden Forschungseinrichtungen für den Bereich Luftverkehrsmanagement. Bei diesen Forschungsaktivitäten werden unter anderem die Daten aus Luftverkehrssimulationen verwendet.

Seit 2008 können diese Daten mit Hilfe der institutsinternen Software „Extensible Workflow Management for Simulations (EWMS)" einheitlich ausgewertet werden. Dem Nutzer dieser Software stehen diverse Auswertungsalgorithmen zur Verfügung, welche nach bestimmten Parametern (z.B. Flugzeit, Verspätung, CO2-Ausstoß, etc.) die Simulationsdaten analysieren. Die Ausgaben werden in einer Tabelle und einem spezifischen Diagramm abgebildet.

Voraussichtlich im Jahr 2014 will das Institut für Flugführung mit dem EWMS2 eine von Grund auf neu entwickelte Version des EWMS einführen. Hierbei sollen die Benutzerführung und die Möglichkeiten der Software grundlegend überarbeitet werden. Ein Aspekt betrifft die Diagrammdarstellung. Statt bisher ein spezifisches Diagramm für eine Auswertung vorzugeben, soll der Nutzer nun in der Lage sein, aus einer Vielzahl sinnvoll einsetzbarer Diagramme auszuwählen.

Kapitel 2

Grundlagen

2.1 Java

Abbildung 2.1: Das Logo von Java - `http://www.java.com/de/about/`

Für die Programmierarbeiten der Aufgaben wurde die Programmiersprache Java gewählt, da auch das Hauptprogramm in Java geschrieben ist. Java ist eine Programmiersprache und eine Laufzeitumgebung. Die zugrundeliegende Technologie dient als Basis für moderne Programme, wie Dienstprogramme, Spiele und Business-Anwendungen.[1]

Java ist eine Weiterentwicklung der Programmiersprache C. Der Grund für die Entwicklung von Java war es, eine Programmiersprache zu entwickeln, mit der man plattformunabhängige Programme entwickeln kann. Dies wird dadurch ermöglicht, dass der Compiler den Quellcode zuerst in Bytecode übersetzt und dieser dann von der Java-Laufzeitumgebung in einen für den Rechner und das Betriebssystem verständlichen Maschinencode übersetzt wird. Java besteht also aus drei Bestandsteilen, der Programmiersprache, den Entwicklerwerkzeugen, wie zum Beispiel einem Compiler, und der Java-Laufzeitumgebung, die es für die einzelnen Betriebssysteme gibt. Die Entwicklerwerkzeuge sind in Programmierumgebungen wie Eclipse enthalten.

[1] vgl.: Was ist die Java-Technologie und wozu wird sie benötigt?, `http://www.java.com/de/download/faq/whatis_java.xml`

2.2 Die Programmierumgebung Eclipse

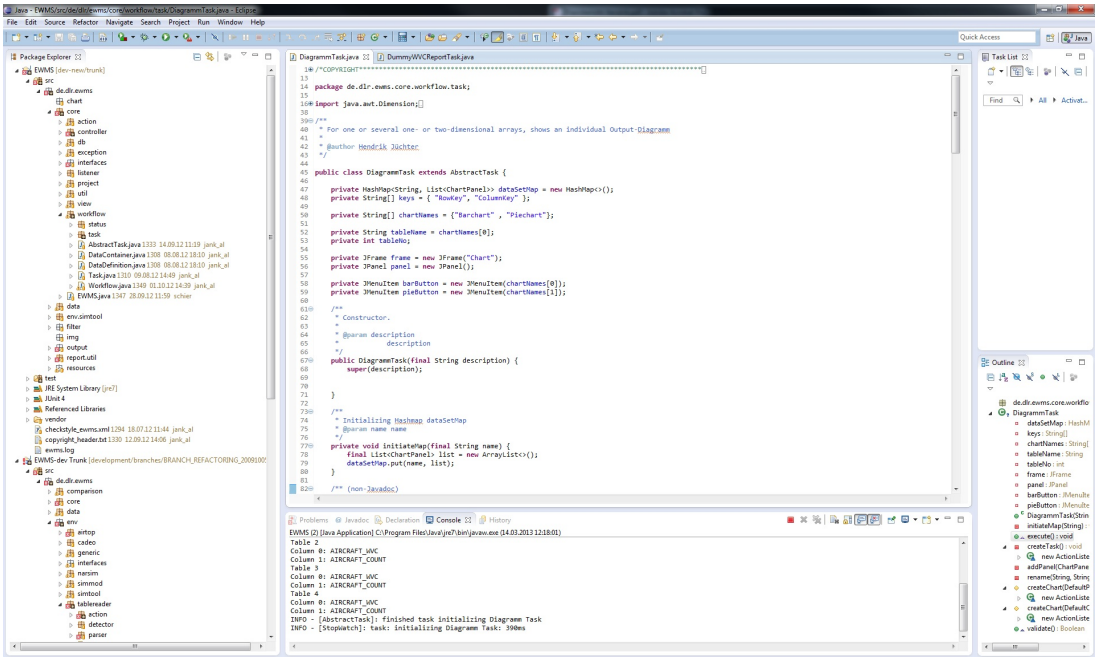

Abbildung 2.2: Die Oberfläche von Eclipse

Eclipse ist das meistgenutzte Programmierwerkzeug zur Programmierung mit Java. Ein wichtiger Bestandteil von Eclipse ist die integrierte Entwicklungsumgebung (eng. Integrated Development Environment (IDE)) für Java. Eclipse ist eine quelloffene Programmierumgebung, um Programme in Java, C++ und vielen weiteren Programmiersprachen, zu entwickeln.

Einige Vorteile und bei Anwendern geschätzte Eigenschaften sind die robuste Entwicklungsumgebung, der inkrementelle Compiler, die automatische Vervollständigung über eine Tastenkombination(STRG + LEER) und die starke Unterstützung von Teamarbeit, sowie die großen Erweiterungsmöglichkeiten durch Plug-Ins.

Die Eclipse Plattform wurde von vornherein als Werkzeug für die Web- und Anwendungsentwicklung entworfen. Die Plattform an sich bringt wenige Endbenutzerfunktionalitäten mit sich. “The value of the platform is what it encourages: rapid development of integrated features based on a plug-in model.”[2] Das bedeutet, die meisten Besonderheiten von Eclipse basieren auf Erweiterungen durch Plug-Ins. Dadurch können die Funktionen schneller entwickelt und zur Plattform hinzugefügt werden, als wenn die Plattform die Besonderheiten von vorn herein unterstützen würde.

Eclipse unterstützt den Programmierer beim Programmieren, indem zum Beispiel der Programmcode farbig dargestellt wird, oder an bestimmten Stellen ein Auswahlmenü an möglichen Befehlen angezeigt wird. Des Weiteren gibt es die Möglichkeit seinen Code auf unsaubere Programmierung mit Hilfe von Checkstyle zu untersuchen. Checkstyle ist ein Open Source Programm zur Überwachung des Quellcodes. Es überwacht die Einhaltung von Code Konventionen. “It automates the process of checking Java code to spare humans of this boring (but important) task.”[3] Mit diesem Satz bringen die Entwickler den Nutzen von Checkstyle auf den Punkt. Checkstyle wird in Eclipse als Plug-In genutzt.

[2]Platform Plug-in Developer Guide. What is Eclipse?, `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fint_eclipse.htm`

[3]Checkytsle 5.6. Overview, `http://checkstyle.sourceforge.net/`

2.3 Das Projekt EWMS

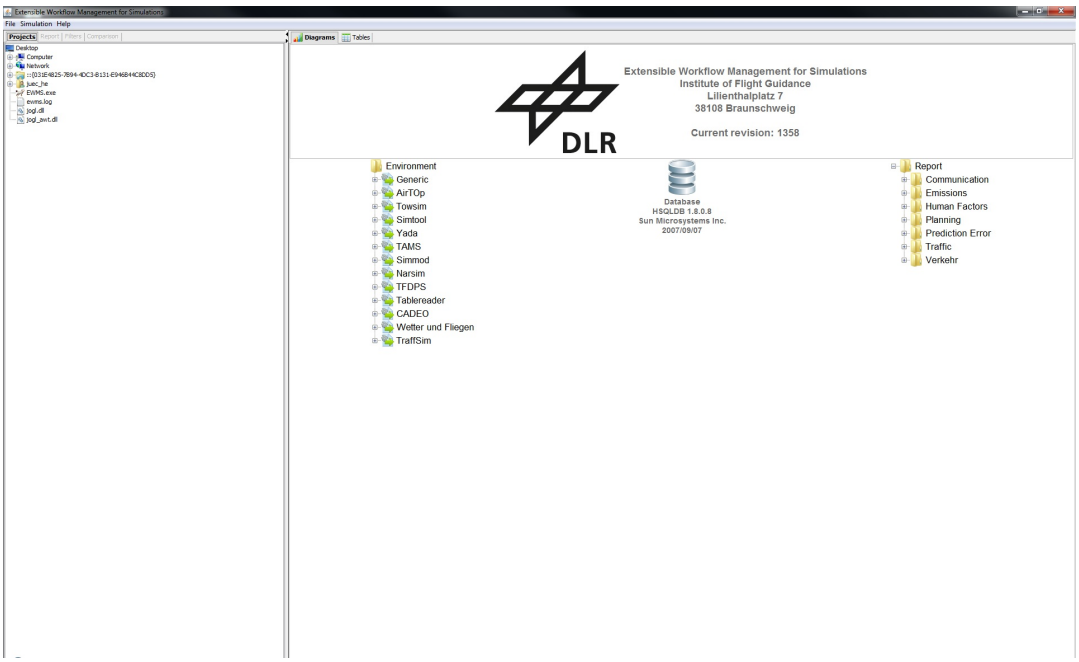

Abbildung 2.3: Die Oberfläche des EWMS

Das Institut für Flugführung nutzt verschiedene Simulationseinrichtungen, deren Auswertungssysteme nicht einheitlich und oft kaum automatisiert sind. Dieser Umstand bedeutet für den Anwender, dass er viel Zeit in die Auswertung, Bewertung und Vergleich der gewonnenen Daten setzen muss. Zur Unterstützung der Simulationsnutzer wurde daher das Softwaresystem Extensible Workflow Management for Simulations (EWMS) entwickelt. Das EWMS ermöglicht es, Simulationsdaten unabhängig von der verwendeten Simulationsumgebung mit einheitlichen Verfahren zu bewerten. Verschiedene Simulationssysteme und auch analytische Werkzeuge können durch eine flexible Architektur schnell und unkompliziert mit dem EWMS verbunden werden. Auch die Auswertungsfunktionalitäten sind flexibel erweiterbar. Die weitgehende Automatisierung verringert erheblich den Aufwand zur Gewinnung nützlicher und gut vergleichbarer Daten. Der Benutzer kann sich damit auf die Interpretation und Bewertung der Ergebnisse konzentrieren.[4]

2.3.1 EWMS 2

Bei dem „Extensible Workflow Management for Simulations 2“ (EWMS 2) handelt es sich um eine Neuentwicklung des EWMS. Die Aufgaben, die das neue EWMS lösen soll, ändern sich nicht grundlegend. Die Anforderungen an die Neuentwicklung sind eine höhere Geschwindigkeit des Programms, größere Möglichkeiten bei den Auswertungen mit weniger Programmieraufwand durch Baukastenprinzip und bessere Übersichtlichkeit.

Die Geschwindigkeitsoptimierungen entstehen mit einer schlankeren Programmierweise, neu programmierten Programmteilen, die vom alten EWMS übernommen werden und Parallelisierung von Programmabläufen. Die schlankere Programmierweise wird durch Behebung der

[4] vgl. Scharnweber, A. , Schier, S. (2010): Das „Extensible Workflow Management for Simulations“ im Einsatz. Abstract

Architekturüberlastung und dem Entfernen von Legacy Code erreicht. Dazu werden Programmteile weggelassen, bei denen sich keine sinnvolle Nutzung herausgestellt hat. Um die Qualität des Programmcodes zu verbessern und die Geschwindigkeit zu verbessern wurde Refactoring angewendet. "Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."[5] Bei Refactoring wird bestehender Programmcode so rekonstruiert, dass die innere Struktur des Codes verbessert wird, ohne das Ergebnis des ursprünglichen Codes zu ändern. Bei der Neuentwicklung eines Programms weiß man von vorn herein, welche Fehler man bei der ersten Entwicklung gemacht hat und was man besser machen kann um das Programm performanter zu gestalten. Außerdem wird mit einer Parallelisierung durch Multithreading-Workflow-Frameworks gearbeitet. Dies ändert den sequentiellen Programmablauf des alten EWMS zu einem parallelisierten Programmablauf beim EWMS 2.

Beim Baukastenprinzip des EWMS 2 handelt es sich um die Auswahl verschiedener Programmbausteine zum Aufbau einer Auswertung. Die einzelnen Programmbausteine laufen jeweils in einem eigenen Task. Bei der ersten Generation des EWMS wird jede Auswertung einzeln programmiert und jedes Auswertungsmodul hat seine eigene Ausgabe, die erzeugt wird. Diese Ausgabe aber ähnelt sich bei allen Auswertungen, sodass man diese in ein Ausgabemodul für Diagramme packen kann. So wird ein Programmbaustein nicht mehr mehrmals gleich entwickelt, sondern einmalig. Das EWMS der zweiten Generation wird dahingehend entwickelt, dass sich der Benutzer des Programms am Anfang verschiedene Bausteine zu seiner individuellen Auswertung zusammensetzt. Die Voraussetzung für das Baukastenprinzip sind einheitliche Eingabe- und Ausgabedaten der Programmbausteine. Dieses System der Bausteine mit einheitlichen Übergabedaten ermöglicht somit eine flexiblere und kompaktere Programmstruktur mit größeren Möglichkeiten.

Die bessere Übersichtlichkeit spiegelt sich in zwei Bereichen wieder, einmal in dem Programmcode selber und einmal in der Gestaltung der Programmoberfläche. Beides wird mit Hilfe der Tasks und den Bausteinen erreicht. Dadurch, dass es verschiedene Bausteine gibt, die in Tasks gestartet werden, gibt es allgemein weniger Programmcode und die Aufgaben, die in die Bausteine programmiert wurden, sind nun einfacher zu finden, da klar strukturiert ist, in welchem Baustein welche Aufgabe zu finden ist. Das alte EWMS hat mehrere Aufgaben in einem Programmteil, wie einer Auswertung. Das neue EWMS baut sich aus mehreren Programmbausteinen zusammen, die jeweils nur eine Aufgabe ausführen, dafür aber von mehreren anderen Bausteinen genutzt werden kann. Die Gestaltung der Programmoberfläche ist übersichtlicher geworden, da nicht mehr aufgelistet werden muss, welche Auswertungen alle zur Verfügung stehen, sondern man sich die Auswertungen nach eigenen Wünschen zusammenstellen kann. Des Weiteren können nun mehrere Tasks in Registerkarten dargestellt werden, sodass das Programm nicht für jede Aufgabe neu initialisiert werden muss.

[5] Fowler, M. : What is Refactoring?, `www.refactoring.com`

2.4 JFreeChart

JFreeChart ist eine freie Diagrammbibliothek für die Programmiersprache Java. Es wurde für die Verwendung von Diagrammen in Programmen, Applikationen, sowie Webanwendungen client- und serverseitig entwickelt. [6]

Mit Hilfe von JFreeChart lassen sich die verschiedensten Diagramme graphisch darstellen. Zum Beispiel Kuchendiagramme, Balkendiagramme oder Liniendiagramme und noch viele weitere. Des Weiteren bietet die Bibliothek unterschiedliche Einstellungsmöglichkeiten für die einzelnen Diagramme und auch die Möglichkeit, das Design der Abbildungen stark zu variieren. Als Beispiel kann man ein Stacked Barchart, ein aufeinander gestecktes Balkendiagramm erzeugen, bei dem die Werte angezeigt werden, sobald man mit der Maus über einen Balken fährt. Dieses lässt sich sowohl in 2D als auch in 3D darstellen, liegend oder stehend und in verschiedenen Farbvariationen.

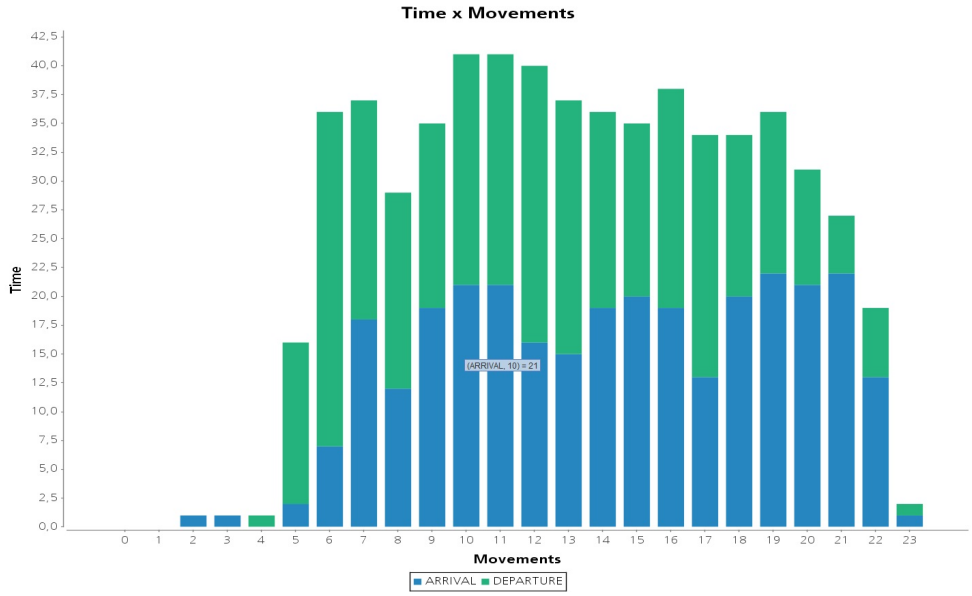

Abbildung 2.4: Beispieldiagramm

In der Abbildung sieht man ein Stacked Barchart beispielhaft für den Einsatz von JFreeChart. In der Mitte sieht man die Daten für einen der Balken, nachdem man mit der Maus über den Balken gefahren ist. Unter dem Diagramm sieht man eine Legende, wofür die verschiedenen Farben im Diagramm stehen.

[6] vgl. Gilbert, D. (2009): The JFreeChart ClassLibrary. Version 1.0.13. Developer Guide. Kapitel 1.1.1

Kapitel 3

Das Diagrammauswahlmodul

Das Modul ist eine personalisierbare dynamische Diagrammauswahl für die Auswertungssoftware „Extensible Workflow Management for Simulations (EWMS)“ in der zweiten Version. Es dient zur Darstellung von Diagrammen. Das Modul sucht im ersten Schritt ein verfügbares Diagramm für die Eingangsdaten selbst aus. Die Auswahlkriterien sind das Klickverhalten des Benutzers und welche Diagrammtypen zu den Daten passen.

Eine weitere Möglichkeit, ein geeignetes Diagramm auszuwählen, ist eine Übersicht über alle möglichen Diagramme. In dieser Übersicht werden die Diagramme nebeneinander dargestellt und der Benutzer kann nun manuell auswählen, welches Diagramm er wählen möchte. Außerdem ist es möglich zwischen verschiedenen Tabellen, die in den Eingangsdaten vorhanden sind zu wechseln.

Der Benutzer ist der Nutzer des Computers. Das Modul zur Diagrammauswahl bietet auch die Möglichkeit, die Nutzerdaten anderer Nutzer zu verwenden, sprich den Nutzer des Moduls zu wechseln.

Die Diagramme werden in Registerkarten organisiert, sodass man schnellen Zugriff auf die verschiedenen Eingangstabellen hat, aus denen die Diagramme erzeugt werden. Es werden nur die Registerkarten erzeugt, die den vorher ausgewählten Diagrammtyp und Eingangstabelle enthalten.

Die Diagrammauswahl wird von Anfang an mit den gegebenen Schnittstellen des EWMS 2 entwickelt, sodass die Anwendung nicht immer neu angepasst werden muss.

3.1 Nutzung existierender Schnittstellen

Existierende Schnittstellen nutzt das Diagrammmodul an möglichst vielen Stellen. So sind die Eingangsdaten für das Programmmodul die Ausgangsdaten anderer Programmmodule. Diese Eingangsdaten sind einheitlich strukturiert, sodass die Umwandlung der Daten in Diagramme generisch programmierbar ist und nicht mehr für verschiedene Daten verschiedene Ausgaben programmiert werden müssen, wie es beim alten EWMS der Fall war.

Außerdem nutzt das Diagrammmodul die gleiche Oberfläche wie das Hauptprogramm, sodass die Übersicht gewährleistet bleibt. Das EWMS 2 wird mit Tasks arbeiten, ähnlich denen eines gängigen Webbrowsers. Mit dem frühen Stand der Entwicklung des EWMS 2 steht am Ende der dritten Praxisphase die Oberfläche mit den Tasks noch nicht zur Verfügung und die Diagramm-Auswahl kann diese Schnittstelle noch nicht nutzen.

Eine dritte vorhandene Schnittstelle ist das Corporate Design für die Diagramme. Das Corporate Design ist einheitlich vorgeschrieben für alle Präsentationen und Arbeiten des DLR. Aus diesem Grund bedient sich das Diagrammmodul einer Vorlage, die als Enumeration für das gesamte EWMS 2 verfügbar ist.

3.1.1 Eingangsdaten

Die Eingangsdaten für die Diagramme bestehen aus den berechneten Daten für die Darstellung der Diagramme sowie aus den Beschreibungen für die Daten. Diese Beschreibungen sind zum Beispiel der Flugtyp oder die Flugphase. Bevor eine Tabelle dargestellt werden kann, müssen die Positionen der Zusatzinformation herausgefiltert werden, damit nur zur Darstellung relevante Daten verwendet werden.

3.2 Transformation von Daten in geeignete Diagramme

Bevor aus den Eingangsdaten darstellbare Diagramme werden, müssen die Daten zur Erstellung der Diagramme aufbereitet werden. Um ein Diagramm zu erzeugen braucht man einen Dataset bestehend aus den Werten, die dargestellt werden, den Vergleichswerten und gegebenenfalls verschiedenen Wertebereichen. Ein Dataset ist eine Menge von Daten. Es ist eine JFreeChart-Datenstruktur zum Speichern der Daten für die Diagrammerzeugung. Es wird in unterschiedlichen Strukturen erstellt, je nachdem für welchen Diagrammtyp es die Daten sammeln soll.

3.2.1 Spalten mit Zahlen herausfiltern

Um diese Datasets zu erzeugen, werden zuerst die Spalten mit Werten aus den Tabellen herausgefiltert. Dazu gibt es die Methode *checkColumn*, die Eingangsdaten übergeben bekommt und eine Liste zurückgibt, die die herausgefilterten Spaltennummern der Spalten mit den passenden Werten beinhaltet.

```
private List<Integer> checkColumn(final DataContainer data) {
    //Rückgabeliste
    final List<Integer> colCheck = new ArrayList<>();
    //Überprüfung für jede Spalte auf Zahlen als Inhalt
    for (int col = 0; col < data.getColumnCount(); col++) {...}
    return colCheck;
}
```

Die folgende Funktion wird für jede Spalte der Eingangstabelle ausgeführt. In ihr wird überprüft, ob die Werte der Spalte aus Zahlenwerten besteht und nicht aus Text. Außerdem müssen noch zwei Datentypen herausgefiltert werden, die intern im EWMS als Zahlen abgespeichert werden, aber Beschreibungen und keine Ergebniswerte sind.

```
final IFDataType colDataDef = data.getDataDefinition().getMetadata()[col];
final String check = colDataDef.getName();
final String columnDataIsNumber = data.getColumn(col)[0].toString();
if (!check.endsWith("ID")
        && !check.endsWith("WVC")
        && columnDataIsNumber.matches("\\d+([.]{1}\\d+)?")) {
    keys[0] = colDataDef.getDescription();
    colCheck.add(col);
}
if (check.endsWith("TYPE")) {
    checkType = true;
}
```

Abschließend überprüft die Funktion, ob der Name eines Datentyp mit *TYPE* endet. Wenn dies zutrifft muss eine Wertbeschreibung bei einem Kuchendiagramm verändert werden.

3.2.2 Vorbereitung der Transformation von Daten

Nachdem die Position der Werte bekannt ist, können die Datasets für die verschiedenen Diagrammtypen erstellt werden. Es gibt drei Methoden für unterschiedliche Mengen an Wertspalten. Es gibt die Fälle, dass es genau eine Spalte mit berechneten Werten gibt, mit genau zwei solchen Spalten, oder mit mehr als zwei Wertspalten. Die ersten beiden Fälle werden für jede Tabelle in den Eingangsdaten ausgeführt und der dritte Fall für den gesamten Datensatz, da dieser bei diesem Fall anders behandelt werden muss.

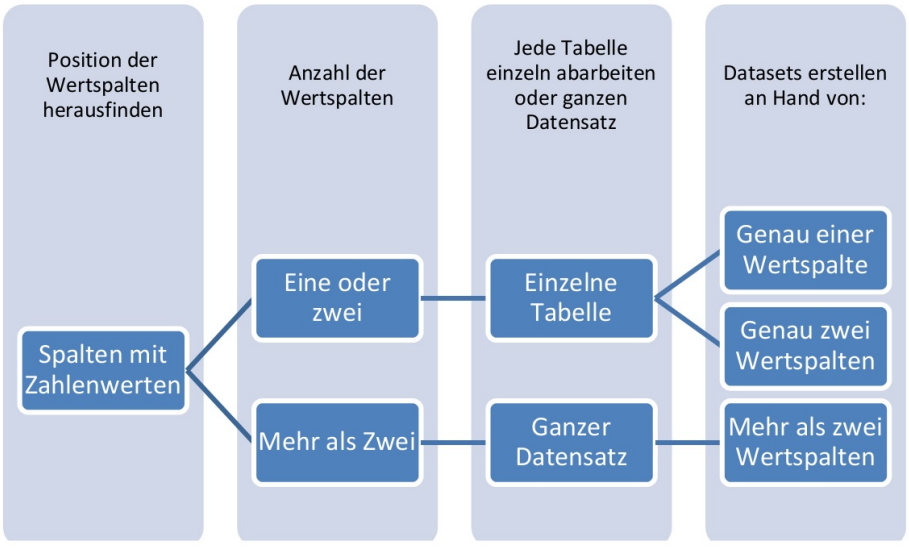

Abbildung 3.1: Entscheidungsdiagramm für die Bearbeitung der Wertspalten

Bevor die Datasets erstellt werden können, müssen diese erst initialisiert werden. Die Datencontainer, in die die Daten gespeichert werden, sind global deklariert, damit diese in der

gesamten Klasse zur Verfügung stehen, und lokal initialisiert. Zwei der Datencontainer werden für alle Tabellen initialisiert. Diese sammeln die Daten für Diagramme ein, auf denen die Inhalte mehrerer Tabellen zu sehen ist, um diese zu vergleichen. Es sind eine *XYSeriesCollection* und ein *DefaultCategoryDataset*. Eine *XYSeriesCollection* entspricht einer Zusammenstellung von *XYSeries*-Objekten und kann als Dataset verwendet werden. Ein *XYSeries*-Objekt sammelt die Daten für jede einzelne Tabelle ein. Ein *DefaultCategoryDataset* ist ein Standard Dataset. Dieser Datasettyp dient in diesem Programm zur Bereitstellung der Daten für Balkendiagramme. Der *DefaultCategoryDataset stackedBarDataSet* sammelt die Daten mehrerer Tabellen, damit daraus ein zusammengestecktes Balkendiagramm entsteht. Die Datasets *pieDataSet* und *barDataSet*, sowie das *XYSeries*-Objekt *lineSeries* werden für jede Tabelle einzeln neu initialisiert und verarbeitet.

```
lineSeriesCollection = new XYSeriesCollection();
stackedBarDataSet = new DefaultCategoryDataset();
List<Integer> colIsNo = null;
for (DataContainer data : getInputs()) {
    pieDataSet = new DefaultPieDataset();
    barDataSet = new DefaultCategoryDataset();
    lineSeries = new XYSeries(data.getData()[0][0].toString());
...
```

Mit der *for*-Schleife iteriert die Methode mit dem Befehl *getInputs()* über alle Eingangstabellen und erstellt einen Datencontainer *data* mit einer Tabelle für den aktuellen Schleifendurchlauf. Die Methoden zur Erstellung der Datasets bekommen die Tabelle Beziehungsweise die gesamten Eingangsdaten und die Liste mit den Wertspalten übergeben.

```
...
    //Herausfiltern der Spalten
    colIsNo = checkColumn(data);
    //Beschriftung X-Achse
    keys[0] = data.getDataDefinition()
        .getMetadata()[colIsNo.get(0)].getDescription();
    //Datasets bei 2 Wertspalten
    if (colIsNo.size() == 2) {
        //Beschriftung Y-Achse
        keys[1] = data.getDataDefinition()
            .getMetadata()[colIsNo.get(1)].getDescription();
        createChartsTwo(data, colIsNo);
    }
    //Datasets bei 1 Wertspalte
    else if (colIsNo.size() == 1) {
        createChartsOne(data, colIsNo);
    }
    ...     //Diagramme erzeugen
}
//Datasets bei vielen Wertspalten
if (colIsNo.size() >= 3) {
    createSeriesChart(getInputs(), colIsNo);
}
...     //weitere Diagramme erzeugen
```

3.2.3 Erstellung der Datasets anhand einer Spalte mit Werten

Beim ersten Fall werden die Daten mit der Methode *createChartsOne* erstellt. Diese Methode baut die Datasets für Kuchen- und Balkendiagramme auf. Zuerst werden die Beschreibungen der Datentypen einer Tabelle für die Beschriftung der Diagramme. Anschließend wird in einer *for*-Schleife über jede Reihe in der Eingangstabelle iteriert und der Wert aus der Wertspalte ausgelesen, sowie die Beschreibung des Wertes aus der ersten Spalte des Diagramms.

Nun wird überprüft, ob die Beschreibung der Werte für das Kuchendiagramm geändert werden muss. Wenn dies zutrifft, besteht die neue Beschreibung aus den Werten der ersten beiden Spalten der Tabelle. Wenn keine Änderung nötig ist, sind die Beschreibungen für die Werte bei Kuchen- und Balkendiagramm gleich.

Nachdem die Werte und Beschreibungen einer Reihe festgelegt sind, werden sie den Datasets *pieDataSet* (Dataset für Kuchendiagramm) und *barDataSet* (Dataset für Balkendiagramm) hinzugefügt. Das Balkendiagramm benötigt darüber hinaus auch noch die Beschreibung des Wertebereichs. Ein Wertebereich wären zum Beispiel Äpfel bei einem Obsthändler, der Wert wäre der Preis und die Beschreibung des Wertes die Art der Äpfel.

```
private void createChartsOne(final DataContainer data,
    final List<Integer> colIsNo) {
    ...
    for (Object[] object : data.getData()) {
        //Ein Wert für die Diagramme
        final double value = Double.parseDouble(object[colIsNo.get(0)]
            .toString());
        //Beschreibung für den Wert
        String compPie = "" + object[0];
        final String compBar = compPie;
        //Beschreibungsanpassung bei bestimmten Datentypen
        if (checkType) {
            compPie = compPie + " " + object[1];
        }
        if (value > 0) {
            String rowKey = object[colIsNo.get(0) - 1].toString();
            if (colIsNo.get(0) == 1) {
                rowKey = keyNames[1];
            }
            pieDataSet.setValue(compPie, value);
            barDataSet.addValue(value, rowKey, compBar);
        }
    }
}
```

Die *for*-Schleife nimmt beim Iterieren über die Daten eine Zeile aus der Datentabelle heraus und stellt für den Schleifendurchlauf diese Zeile als Objekt-Array bereit. Um nun einen *double*-Wert aus dem Array auslesen zu können, wird mit der Liste der Wertspalten das benötigte Feld aus dem Array herausgenommen. Anschließend wird dieser zu einem String umgewandelt und aus diesem String ein *double*-Wert ausgelesen.

```
final double value = Double.parseDouble(object[colIsNo.get(0)].toString());
```

Der vorliegende Wert kann in jedem Fall zu einem Double umgewandelt werden, da in der Liste *colIsNo* die Spaltennummern abgespeichert sind, in denen nur Zahlen vorkommen.

3.2.4 Erstellung der Datasets anhand zweier Spalten mit Werten

Wenn 2 Spalten Zahlenwerte beinhalten, wird die Tabelle mit der Methode *createChartsTwo* verarbeitet. Das Prinzip hier folgt dem Prinzip der ersten Methode, der *createChartsOne*. Es wird wieder mit Hilfe einer *for*-Schleife über jede Reihe in der Eingangstabelle iteriert, sodass die Reihe in einem Objektarray zwischengespeichert werden kann. Danach werden dann die beiden Zahlenwerte aus den richtigen Spalten in der Reihe ausgelesen.

```
private void createChartsTwo(final DataContainer data,
            final List<Integer> colIsNo) {
    for (Object[] object : data.getData()) {
        final double value = Double.parseDouble(object[colIsNo.get(1)]
            .toString());
        double compValue = Double.parseDouble(object[colIsNo.get(0)]
            .toString());
        String comp = "" + compValue;
```

Nun gibt es hier eine *if*-Abfrage, in der überprüft wird, ob die Werte der Spalte mit den zu vergleichenden Daten vom Typ Zeit sind. Wenn dies der Fall ist, werden die Zeitwerte von Sekunden in Stunden umgerechnet.

```
        if (data.getDataDefinition().getMetadata()[colIsNo.get(0)]
                .getName().equals("TIME")) {
            compValue = compValue / 60 / 60;
            comp = "" + (int) compValue;
        }
```

Nachdem die Werte zur Verfügung stehen, werden diese den Datasets und der Lineseries übergeben. Die beiden Datasets *barDataSet* und *stackedBarDataset* werden gleich für jede Tabelle mit den gleichen Werten befüllt, doch wird im gesamten Prozess zur Erstellung der Datasets das *stackedBarDataset* nur einmal initialisiert und das *barDataSet* für jede Tabelle. Nach der Schleife wird noch die Lineseries der Collection hinzugefügt.

```
        lineSeries.add(compValue, value);
        pieDataSet.setValue(comp, value);
        barDataSet.addValue(value, object[0].toString(), comp);
        stackedBarDataSet.addValue(value, object[0].toString(), comp);
    }
    lineSeriesCollection.addSeries(lineSeries);
}
```

Aus *lineSeriesCollection* werden zwei verschiedene Diagrammtypen erzeugt. Einmal ein Liniendiagramm und ein Balkendiagramm. Beide Diagrammtypen arbeiten mit XY-Koordinaten und können mit einem Zeitablauf als X-Achse ausgestattet werden. Das folgende Liniendiagramm enthält eine Zeitachse und beinhaltet Eingangsdaten aus zwei Tabellen.

Abbildung 3.2: Ein Liniendiagramm mit Zeitachse

3.2.5 Erstellung der Datasets anhand vieler Spalten mit Werten

Falls die Eingangstabellen mehr als zwei Spalten mit Zahlenwerten enthalten, wird die *createSeriesChart* ausgeführt. Diese Methode wird für die gesamten Eingangsdaten aufgerufen und nicht nur für eine Tabelle. Dies wird gemacht, da hier die Einträge der Datasets nicht aus einer ganzen Reihe der Tabelle erzeugt werden, sondern aus zwei Werten der Reihe, sprich zwei Spalten der Tabelle. Hierzu wird die erste Spalte mit Zahlenwerten als Spalte mit Vergleichswerten genommen und die anderen Spalten jeweils als Spalte mit Hauptwerten. Außerdem werden die Beschreibungen der Daten einer Tabelle benötigt.

Die Methode startet mit einer normalen *for*-Schleife, die durchläuft, bis ein bestimmter Wert erreicht wird. In diesem Fall ist das die Anzahl der Eingangstabellen. In der Schleife wird dann als erstes die Tabelle ausgelesen und als 2-dimensionales Objektarray zwischengespeichert. Anschließend wird die *DataDefinition*, in der die Beschreibung der Daten gespeichert sind, ausgelesen.

```
private void createSeriesChart(final Vector<DataContainer> vectorData,
        final List<Integer> colIsNo) {
    //Jede Tabelle einzeln bearbeiten
    for (int tableCount = 0; tableCount < vectorData.size(); tableCount++) {
        //Tabelle auslesen
        final Object[][] data = vectorData
            .get(tableCount).getData();
        //Beschreibungen der Daten auslesen
        final DataDefinition definition = vectorData
            .get(tableCount).getDataDefinition();
        ... //Spalten bearbeiten
    }
}
```

Im nächsten Schritt wird eine weitere *for*-Schleife erstellt, die so oft durchläuft, wie eine Tabelle Spalten enthält. In der Schleife wird dann als erstes die Beschreibung der Spalte ausgelesen. Wenn der Inhalt der Spalte aus Zahlenwerten besteht und die Beschreibung der Spalte

nicht *Time* lautet, also die Daten nicht vom Typ Zeit sind, werden die Hauptwerte aus der Spalte ausgelesen. Nun wird eine neue *XYSeries* mit der Datenbeschreibung der Spalte initialisiert und der Collection hinzugefügt, damit die Werte an gewünschter Position abgespeichert werden.

```
//Jede Spalte bearbeiten
for (int colIndex = 0; colIndex < colIsNo.size(); colIndex++) {
    //Spaltenbeschreibung auslesen
    final String label = definition.
        getMetadata()[colIndex].getDescription();
    //Spalte mit Zahlenwert und ohne Zeiten weiterverarbeiten
    if (colIsNo.contains(colIndex) && !label.equals("Time")) {
        //Die XYSeries für die aktuelle Spalte erstellen
        final XYSeries series = new XYSeries(label);
        //XYSeries der Collection hinzufügen
        lineSeriesCollection.addSeries(series);
        ... //Reihen bearbeiten
        //Beschriftung für Diagramm
        keys[0] = definition.getMetadata()[0].getName();
        keys[1] = label;
    }
}
```

Im nächsten Schritt werden die bereits initialisierten *XYSeries* mit Werten gefüllt. Dazu wird eine weitere *for*-Schleife verwendet, die die Anzahl der Reihen als Index enthält. In der Schleife werden die beiden Werte für die *XYSeries* ausgelesen. Das sind zum einen der Wert der aktuellen Reihe aus der ersten Spalte mit Zahlenwerten als X-Wert, egal ob Wert vom Typ Zeit oder nicht, und zum anderen der Wert der aktuell durchlaufenen Spalte als Y-Wert. Um die Werte der richtige *XYSeries* zuzuweisen, wird diese mit Hilfe der Spaltenbeschreibung aus der Collection herausgenommen. Danach werden die beiden Werte hinzugefügt oder erneuert, wenn ein X-Wert schon in der *XYSeries* enthalten ist.

```
//Jede Reihe bearbeiten
for (int rowIndex = 0; rowIndex < data.length; rowIndex++) {
    //X und Y Wert auslesen
    final Double time = Double.parseDouble(""
        + data[rowIndex][colIsNo.get(0)]);
    final Double value = Double.parseDouble(""
        + data[rowIndex][colIndex]);
    //XYSeries heraussuchen und Werte hinzufügen oder erneuern
    lineSeriesCollection.getSeries(label).addOrUpdate(time, value);
}
```

3.3 Die Diagramme

Der wichtigste Teil der personalisierbaren dynamischen Diagrammauswahl sind die Diagramme selber. Diese werden aus den schon beschriebenen Datasets erstellt. Das Diagrammmodul kann zurzeit drei verschiedene Balkendiagrammtypen, Kuchendiagramme oder Liniendiagramme je nach Art der Eingangsdaten erstellen und weitere Diagrammtypen können mit wenig Aufwand hinzugefügt werden. Die Diagramm die das Programmmodul erstellt, werden einmalig pro Modulaufruf erzeugt und werden zwischengespeichert, sodass sie immer wieder angezeigt werden können, ohne die Diagramme neu zu erzeugen.

3.3.1 Datenstruktur Diagrammzwischenspeicher

Um die Diagramme jederzeit abrufbereit zu haben, sind sie in Listen abgespeichert, die in einer Hashmap organisiert sind. In einer Liste befinden sich immer nur die Diagramme eines Diagrammtyps. Diese Listen sind dann mit der Bezeichnung des Diagrammtyps als Schlüssel in der Hashmap *dataSetMap* abgespeichert.

```
private HashMap<String, List<JFreeChart>> dataSetMap = new HashMap<>();
```

Das Diagrammmodul unterstützt Balkendiagramme, Kuchendiagramme, Liniendiagramme, ineinandergeschachtelte Balkendiagramme und achsenorientierte Balkendiagramme. Die Bezeichnungen werden in dem global verfügbaren Stringarray *chartNames* abgespeichert.

```
private String[] chartNames = {"Barchart" , "Piechart" , "Linechart"
    , "Stackedbarchart", "XYBarchart"};
```

Sobald das Diagrammmodul gestartet wird, wird im Konstruktor die Methode *initiateMap* aufgerufen. Diese Methode fügt der Hashmap *dataSetMap* die Listen hinzu, sodass die Diagramme später durch einen einfachen Aufruf der Hashmap gespeichert und wieder abgerufen werden können. Die Methode besteht aus einer *for*-Scheife, dessen Durchlaufzahl der Länge des Stringarrays *chartNames* entspricht, also der Anzahl an Diagrammtypen. Bei jedem Schleifendurchlauf wird eine neue Liste erstellt, in der Diagramme gespeichert werden können, die mit JFreeChart erstellt wurden. Diese Liste wird dann mit der Typbezeichnung eines Diagramms als Schlüssel in der Hashmap abgelegt.

```
//Initialisierung der Hashmap dataSetMap
private void initiateMap() {
    //für jeden Diagrammtyp
    for (int initNo = 0; initNo < chartNames.length; initNo++) {
        //neue Liste erstellen
        final List<JFreeChart> list = new ArrayList<>();
        //Liste mit Typbezeichnung als Schlüssel zur Hashmap hinzufügen
        dataSetMap.put(chartNames[initNo], list);
    }
}
```

Damit das Diagrammmodul weiß, welcher Diagrammtyp und welche Tabelle aktuell ausgewählt sind, existieren die beiden Variablen *diagrammName* und *tableNumber*. Der String *diagrammName* hat den ersten Eintrag aus *chartNames* als Defaultwert.

```
private String diagrammName = chartNames[0];
private int tableNumber;
```

Das Merken von Diagrammtyp und Tabellennummer ist für das gesamte Programmmodul wichtig, denn bei jeder Darstellung eines Diagramms werden diese beiden Variablen als Schlüsselwert verwendet. Gesetzt werden diese jedes Mal, wenn eine neue Tabelle beziehungsweise ein neuer Diagrammtyp ausgewählt wurde.

3.3.2 Diagrammerzeugung

Die Diagramme werden mit Hilfe von vier Methoden erstellt, die alle die gleiche Bezeichnung haben aber unterschiedliche Übergabetypen. Diese Methoden nennt man dann überladene Methoden. "Eine überladene Methode ist eine Funktion mit dem gleichen Namen wie eine andere Funktion, aber einer unterschiedlichen Parameterliste." [1] Das Verwenden von überladenen Methoden hat den Vorteil, dass man für unterschiedliche Voraussetzungen mit ähnlichem Ziel die gleichen Methodenaufrufe verwenden kann.

Für die Erzeugung eines Diagramms braucht man mehrere Daten. Man benötigt ein Dataset, eine Diagrammbeschreibung und Achsenbeschreibungen. Die Achsenbeschreibungen sind im Diagrammmodul im Array *keys* gespeichert.

```
private String[] keys = { "RowKey", "ColumnKey" };
```

Die beiden Stringwerte, die in *keys* gespeichert werden, beschreiben die X-Achse und die Y-Achse der Diagramme und bilden zusammengesetzt die Beschreibung des Diagramms.

Mit Hilfe der *PlotOrientation* kann man die Ausrichtung des Diagramms zu horizontaler oder vertikaler Ausrichtung ändern. Falls beim Balkendiagramm der Fall eintritt, dass die X-Achse Zeitwerte enthält und keine Wörter, wird die Ausrichtung auf Horizontal gesetzt, anderenfalls ist sie Vertikal. Die Daten der X-Achse können nicht gelesen werden, wenn die Anzahl der Zeichen zu groß ist. Dies ist bei Wörtern meistens der Fall, deshalb wird die Ausrichtung überprüft. Die anderen Diagramme sind horizontal ausgerichtet.

```
PlotOrientation plotOri = PlotOrientation.VERTICAL;
if (!keys[0].equals("Time")) {
    plotOri = PlotOrientation.HORIZONTAL;
}
```

Nachdem alle Bestandteile für die Erstellung der Diagramme vorhanden sind, können diese erstellt werden. Mit Hilfe der JFreeChart-Bibliothek wird nun über die Funktion *ChartFactory* ein Diagramm erzeugt. Beispielhaft benötigt die Erstellung eines normalen oder gestackten Balkendiagramms drei Stringvariablen für die Beschreibung des Diagramms und der Achsen, außerdem ein *categoryDataset*, eine *PlotOrientation* und drei Booleanvariablen für das Anzeigen von Zusatzinformationen im Diagramm.

```
// Diagramm generieren
final JFreeChart barChart = ChartFactory.createStackedBarChart(
    keys[0] + " x " + keys[1] //Beschreibung
    , keys[0]              //X-Achse
    , keys[1]              //Y-Achse
    , dataSetBar           //Dataset
    , plotOri              //Ausrichtung
    , true               //Legende anzeigen
    , true               //Tooltipps anzeigen
    , false);              //URL anzeigen
```

[1] Ullenboom, C. (2012): Java ist auch nur eine Insel. 2.7.9 Methoden überladen

Mit Hilfe der folgenden Programmzeile wird ein Balkendiagramm erstellt und in die richtige Liste in der Hashmap geladen. Dieser Befehl funktioniert mit allen anderen Diagrammtypen genauso, nur müssen die Übergabedaten (chartNames[0] = Barchart; barDataSet) geändert werden.

```
dataSetMap.get(chartNames[0]).add(createChart(barDataSet));
```

In der Programmzeile wird zunächst die gewünschte Liste mit dem Schlüssel *chartNames[x]* aus der Hashmap geladen, in der dann per *.add*-Befehl das neu erstellte Diagramm übergeben wird. Der Befehl *createChart(Dataset)* gibt das fertige Diagramm zurück.

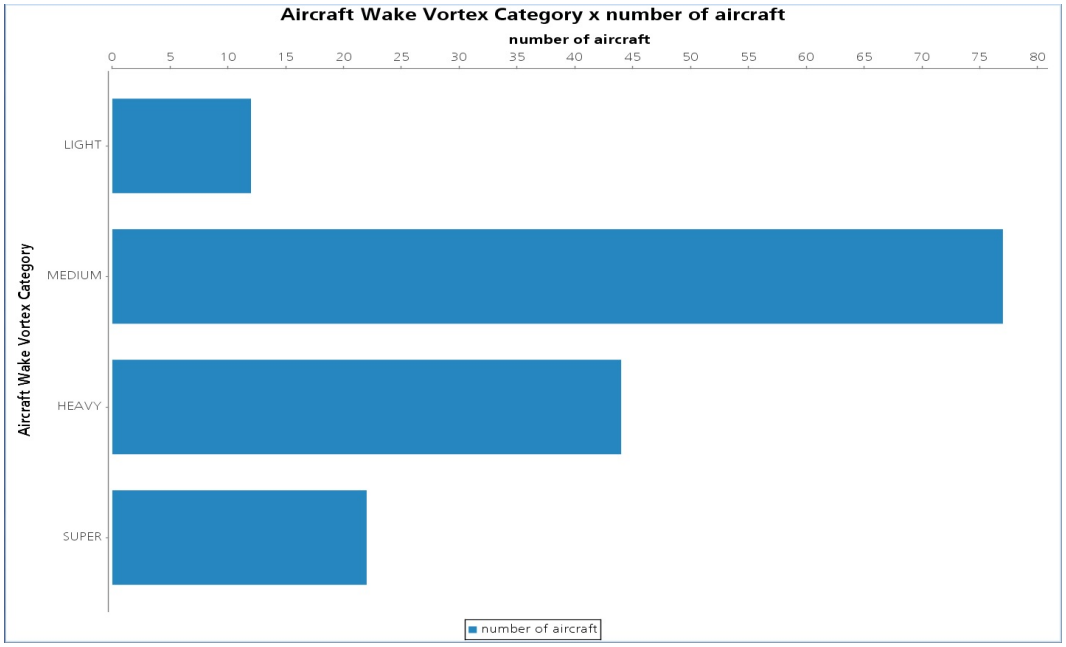

Abbildung 3.3: Ein Balkendiagramm

Für das Abrufen eines Diagramms wird folgender Befehl verwendet:

```
dataSetMap.get(diagrammName).get(tableNumber);
```

Es wird die gewünschte Liste mit Hilfe des Namens des Diagrammtyps aus der Hashmap geladen. Aus dieser Liste wird dann anhand der Tabellennummer das richtige Diagramm geladen. Da die Tabellen sortiert verarbeitet werden, entspricht die Tabellennummer der Position in der Liste.

3.3.3 Besonderheiten bei der Diagrammerstellung

Es gibt einige Besonderheiten und Unterschiede bei der Erstellung unter den einzelnen Diagrammen. Für die Erzeugung eines Kuchendiagramms werden keine Achsbeschreibungen benötigt, da ein Kuchendiagramm rund ist und keine zwei Achsen enthält. Was bei anderen Diagrammtypen als Achsbeschreibung angegeben wird, kann man beim Kuchendiagramm mit der Beschreibung der einzelnen Teilstücke vergleichen.

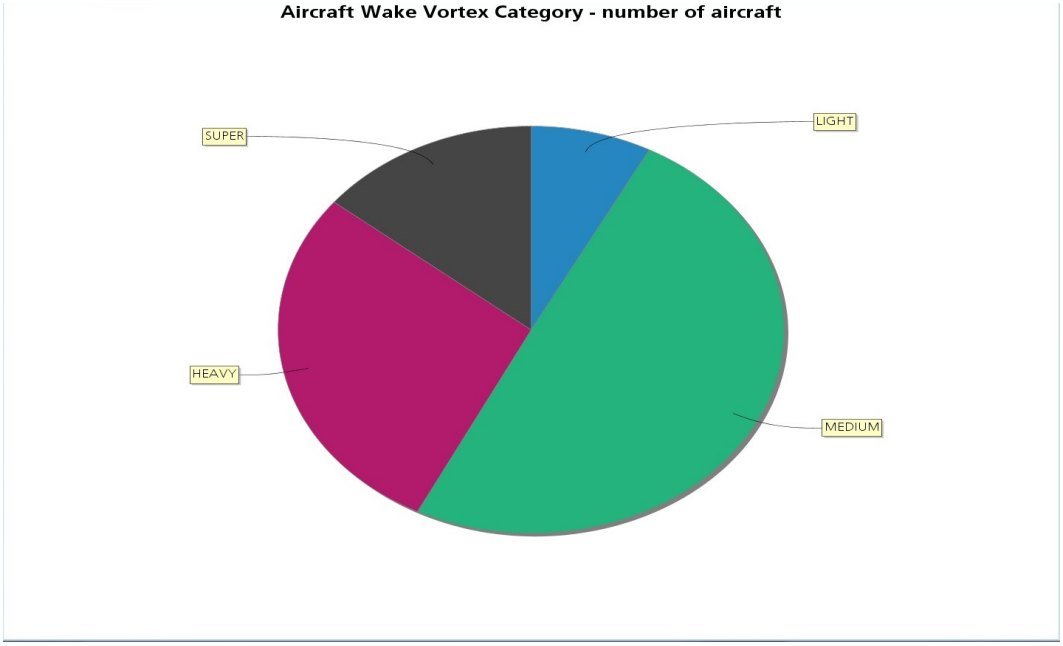

Abbildung 3.4: Ein Kuchendiagramm

Durch die Kreisform des Kuchendiagramms benötigt die *ChartFactory* auch keine Ausrichtung des Diagramms.

Für die Erstellung eines *Stacked Barchart* wird die gleiche Methode verwendet wie bei einem normalen Balkendiagramm. Genau genommen ist das normale Balkendiagramm ein geschachteltes Balkendiagramm mit nur einer Quelle, wodurch keine Schachtelung bei der Ausgabe entstehen kann. Bei JFreeChart gibt es auch eine Möglichkeit ein normales Balkendiagramm zu erstellen, ohne den Umweg über das *Stacked Barchart* zu nehmen, doch in diesem Fall ist die Zusammenlegung sinnvoller.

Bei den XY-Charts *XYBarChart* und *XYLineChart* ist die Besonderheit, das alle Diagramme dieser beiden Typen auf einer Collection basieren. Aus dieser einen Collection werden die einzelnen *XYSeries* ausgelesen und einzeln als Dataset verwendet. Für das *XYBarChart* werden die einzelnen Collections zu XYDataset umgewandelt.

Wenn dann der Fall eintritt, das eine Collection zur Verwendung als Dataset erstellt wurde, werden in einer *for*-Schleife alle *XYSeries* der Collection behandelt. Es wird eine neue Collection erstellt, die eine einzige *XYSerie* bekommt. Diese wird dann auch nochmal in ein neues *XYDataset* gespeichert. Anschließend werden aus beiden Neuerstellungen Diagramme aufgerufen.

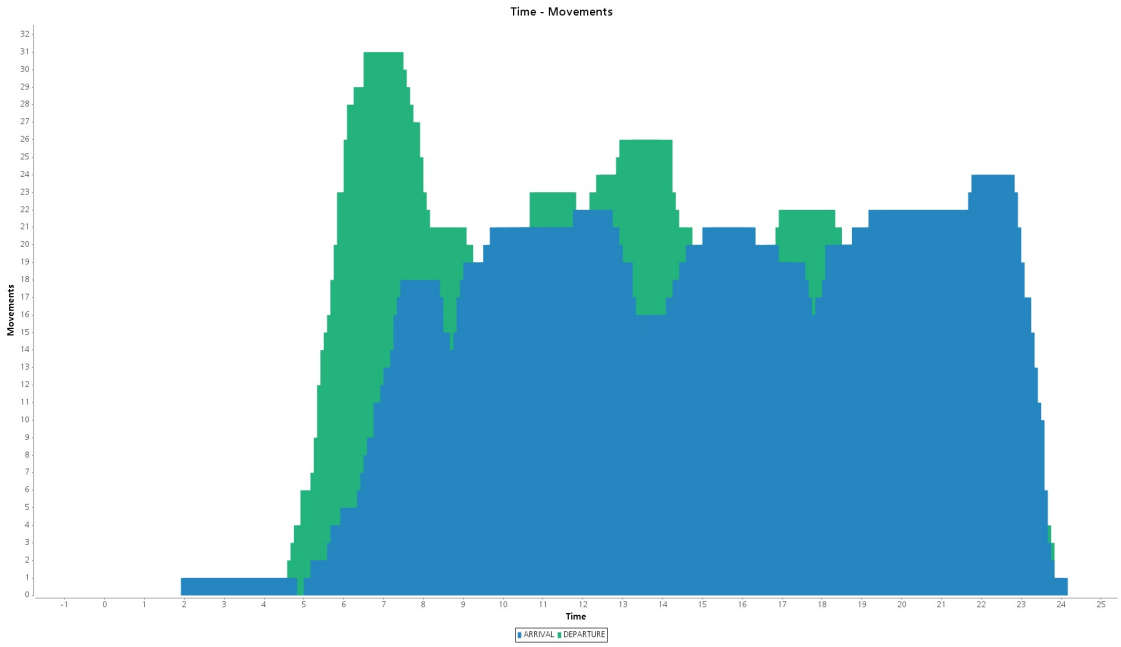

Abbildung 3.5: Ein XYBarchart

```
//Schleife für die Erzeugung von Diagrammen aus einzelnen XYSeries
for (int serieNo = 0; serieNo < lineSeriesCollection
        .getSeriesCount(); serieNo++) {
    //Collection mit einer Serie erstellen
    final XYSeriesCollection newCollection = new XYSeriesCollection(
        lineSeriesCollection.getSeries(serieNo));
    final XYDataset newSingleLineDataSet = newCollection;
    //Diagramme erzeugen
    dataSetMap.get(chartNames[2]).add(createChart(newSingleLineDataSet));
    dataSetMap.get(chartNames[4]).add(createChart(newCollection));
}
```

Vor der Schleife wird zuerst ein *XYDataset* mit allen Daten der Collection erstellt. Nach der Schleife werden auch noch mit den Zusammenfassungen der *XYSeries* Diagramme erzeugt, sodass aus einer Collection somit vier verschiedene Diagrammausgaben erzeugt werden.

```
//XYDataset für XYBarchart erzeugen
final XYDataset lineDataSetColl = lineSeriesCollection;
...
dataSetMap.get(chartNames[2]).add(createChart(lineDataSetColl));
dataSetMap.get(chartNames[4]).add(createChart(lineSeriesCollection));
```

3.4 Manuelle Diagrammauswahl

Das Diagrammmodul bietet eine automatische Diagrammauswahl und eine manuelle Diagrammauswahl. Bei der manuellen Diagrammauswahl werden die möglichen Diagrammtypen in einer Übersicht nebeneinander beziehungsweise übereinander dargestellt. Der Nutzer kann nun durch einen Klick auf den jeweiligen Diagrammtyp jenen für die genauere Ansicht auswählen. Es werden alle einzelnen Tabellen für jeden Diagrammtyp dargestellt. Diese Schachtelung der

Übersicht dient dazu alle Daten in einem Überblick zu sehen. Der Nutzer kann über die Diagrammübersicht für Tabellen, oder über ein Menü zwischen den verschiedenen Tabellen hin und her wechseln.

```
for (int index = 0; index < chartNames.length; index++) {
    if (!dataSetMap.get(chartNames[index]).isEmpty()) {
        //Panel für die Diagramme erzeugen
        final JPanel chartPanel = new JPanel();
        ...
        //Diagramme zum Panel hinzufügen
        chartPanel.add(chart);
    }
}
```

Zuerst wird eine for-Schleife erzeugt, die sich an einer Liste mit den verfügbaren Diagrammtypen orientiert. Die for-Schleife läuft so oft durch, wie es Diagrammtypen gibt. Im zweiten Schritt wird mit einer if-Abfrage abgefragt, ob für den Diagrammtyp, der gerade an der Reihe ist, aufbereitete Daten vorliegen. Beziehungsweise wird überprüft, ob in der Hashmap *dataSetMap* die Position für den abgefragten Diagrammtyp nicht leer ist. Eine Hashmap ist eine Liste, bei der jeder Eintrag mit einem Schlüssel verknüpft wird, ähnlich wie in einer Datenbank. Sind nun aufbereitete Daten vorhanden, wird ein Diagrammpanel erzeugt. Dieses wird am Schluss dem Panel des Tasks übergeben und der nächste Diagrammtyp in der for-Schleife ist dran. Ein Panel ist ein Standardcontainer für Bedienelemente. Er wird in diesem Fall zur Gruppierung der Diagramme genutzt.

Nachdem das Diagrammpanel erstellt wurde, werden die eigentlichen Diagramme erzeugt und dem Diagrammpanel übergeben. Zuerst wird die Anzahl der Tabellen abgefragt. Dies dient der Entscheidung, ob nur ein Diagramm in der Übersicht erzeugt wird, oder eine Übersicht über alle Tabellen. Außerdem dient es der Erstellung eines Rasters für die Anzeige der Diagramme.

```
//Tabellenanzahl
int noOfTables = dataSetMap.get(chartNames[index]).size();
//x-Wert für Raster
final int x = Math.round(noOfTables / 2);
//y-Wert für Raster
int y = noOfTables;
if (checkStacked && (index == 2 || index == 4)) {
    //Herausnehmen der Tabellenzusammenfassung
    noOfTables--;
    //Aktualisierung des y-Wertes
    y = noOfTables;
}
if (noOfTables >= 4) {
    //Ab einer der Anzahl 4 Erzeugung eines gleichmä ß igen Rasters
    y = x;
}
//Raster erstellen
chartPanel.setLayout(new GridLayout(y, x));
```

Als nächstes wird eine neue for-Schleife gestartet, in der die einzelnen Diagramme für die Tabellen erzeugt werden. Diese werden, wenn es nur eine Tabelle in den Eingangsdaten gibt, in Standardgröße dargestellt, ansonsten werden sie verkleinert und in der Diagrammübersicht über Tabellen dargestellt. Wie schon eine Ebene höher geschehen wird nun das *ChartPanel* dem Diagrammpanel übergeben.

Bevor nun das Diagrammpanel an das Hauptpanel übergeben werden kann, fehlt den Diagrammen noch die Möglichkeit es anzuklicken. Dazu wird den Diagrammen je ein *Mouselistener* hinzugefügt. Ein *Mouselistener* ist ein *Actionlistener*. Ein *Actionlistener* überwacht ein Objekt auf eine Aktion die eintreten könnte. Wenn diese Aktion eintritt wird dann ein bestimmter Programmcode ausgeführt. In diesem Fall ist es das Ereignis ein *mouseClicked*, also das klicken der Maustaste auf dem Objekt. Wenn nun mit der Maustaste auf das Diagramm geklickt wurde, wird zuerst der Diagrammtyp für die gesamte Klasse gesetzt, sodass die anderen Methoden damit arbeiten können. Dann werden alle alten Registerkarten entfernt und die Menüleiste des Fensters generiert. Im Anschluss wird die Übersicht über die Tabellen in einem einzelnen Tab für den ausgewählten Diagrammtyp erzeugt, oder ein einziges Diagramm, falls es nur eine Tabelle gibt.

```
chart.addMouseListener(new MouseListener() {
    public void mouseClicked(final MouseEvent arg0) {
        //Diagrammtyp festlegen
        diagrammName = chartNames[chartNo];
        //alte Registerkarten entfernen
        tab.removeAll();
        //Menüleiste neu erzeugen
        createMenuBar();
        //Übersicht über Tabellen erzeugen
        addOrUpdateChartTab();
        //Diagramm der Klickliste hinzufügen
        dynamicUser.addList(diagrammName);
    }
});
```

Abbildung 3.6: Manuelle Diagrammauswahl

Die manuelle Auswahl wird beim Start des Programms aufgerufen, wenn der Benutzer noch keinen Eintrag in der Liste Diagrammklickliste hat, oder die zur Verfügung stehenden Tabellen anhand der Eingangsdaten den am meisten ausgewählten Diagrammtyp nicht enthalten. Des

Weiteren wird mit der manuellen Auswahl der Diagrammtyp gewechselt. Die Diagrammübersicht über Tabellen, die ähnlich aufgebaut ist wie die manuelle Diagrammauswahl steht immer zur Verfügung, sobald es mehr als eine Tabelle in den Eingangsdaten gibt, oder der Diagrammtyp nicht schon alle Tabellen in einem Diagramm verarbeitet.

3.4.1 Vorauswahl des Diagrammtyps

Bevor eine manuelle Diagrammauswahl gestartet werden kann, muss erst überprüft werden, ob eine automatische Diagrammauswahl machbar ist. Dazu wird aus der Klasse *dynamicUser*, die zur Nutzerverfolgung dient, ein Diagrammtyp für den Nutzer ausgewählt.

```
public String getUserDiagramm() {
    //Diagramm ID mit Default-Wert laden
    String diagrammID = diagramm;
    //Überprüfen, ob Nutzer Daten hat
    if (!userMap.get(user).isEmpty()) {
        //Meistgeklickten Typ abfragen
        diagrammID = getMostClicked(userMap.get(user));
    }
    return diagrammID;
}
```

In der Methode *getUserDiagramm* wird ein String *diagrammID* erzeugt und die mit einem vorab veränderten Wert, oder Null, wenn sie vorab nicht festgelegt wurde, belegt. Anschließend wird überprüft, ob der Nutzer schon mit einem Diagramm verknüpft wurde. Wenn dies der Fall ist, wird das Diagramm ausgewählt, das der Nutzer am häufigsten aus der manuellen Diagrammauswahl ausgewählt hat. Ist bisher kein Diagrammtyp manuell ausgewählt worden, wird Null zurückgegeben und daraufhin die manuelle Diagrammauswahl gestartet.

Nachdem ein Diagramm ausgewählt wurde, werden in der Hauptklasse die Bedingungen für eine dynamische Auswahl mit einer if-Abfrage überprüft. Die Bedingungen sind, dass der Nutzer schon mal ein Diagramm ausgewählt hat, dies die erste Diagrammauswahl in diesem Programmdurchlauf ist und ob für den vorab gewählten Diagrammtyp auch Daten in der *dataSetMap* vorliegen. Treffen die Bedingungen alle zu, wird der Diagrammtyp als Diagrammübersicht über alle Tabellen geladen, ansonsten wird die manuelle Diagrammauswahl gestartet.

```
private void createOverview() {
    //Diagrammtyp abrufen
    diagrammName = dynamicUser.getUserDiagramm();
    //Bedingungen prüfen
    if (dynamicUser.userHasDiagramm() && firstChoice
            && !dataSetMap.get(diagrammName).isEmpty()) {
        //Menüleiste erstellen
        createMenuBar();
        //Diagramme automatisch erstellen
        addOrUpdateChartTab();
    } else {
        //Manuelle Diagrammauswahl starten
        manualOverview();
    }
}
```

3.4.2 Diagrammübersicht über Tabellen

Die Diagrammübersicht über Tabellen gibt einen Überblick über alle Tabellen der Eingangsdaten. Sie besteht aus den Diagrammen der einzelnen Tabellen, die in einem Raster angezeigt werden. Man findet Sie in der manuellen Diagrammübersicht vor und in einer eigenen Registerkarte nachdem ein Diagrammtyp ausgewählt wurde. Wie bei der manuellen Diagrammauswahl kann man die einzelnen Diagramme anklicken. Die *Mouselistener* leiten dann weiter zu den einzelnen Diagrammen. Die neu erzeugten Diagramme werden in neuen Registerkarten dargestellt. Der Aufbau des Programmcodes entspricht dem der manuellen Diagrammauswahl.

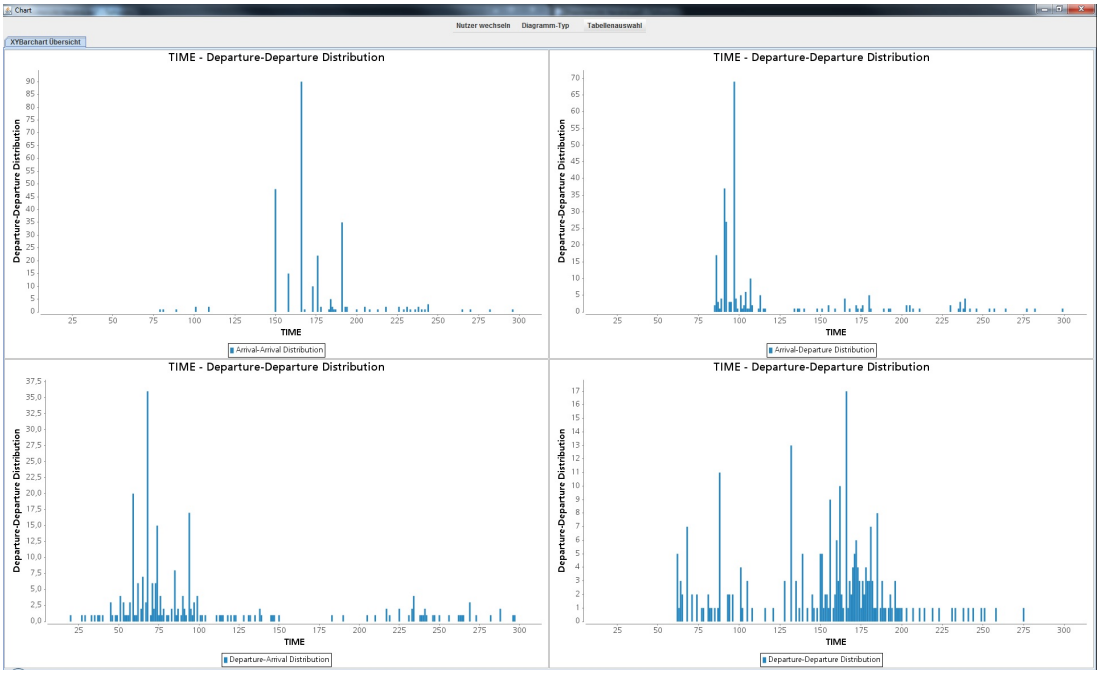

Abbildung 3.7: Diagrammübersicht über Tabellen

3.5 Verfolgung des Nutzerverhaltens

Damit das Diagrammmodul einen Diagrammtyp für den Nutzer personalisiert und automatisch auswählen kann, muss das Nutzerverhalten verfolgt werden. Es wird dazu ein drei dimensionale Liste angelegt, die für verschiedene Nutzer die Häufigkeit der Aufrufe für die Diagrammtypen speichert. Diagrammmodulintern kann der Nutzer gewechselt werden. Die Nutzerverfolgung ist in einer neuen Klasse *DynamicChoise* programmiert. Um auf die Methoden in der Klasse in der Klasse zugreifen zu können, wird sie in der Hauptklasse als *dynamicUser* aufgerufen und initialisiert. Nun kann man mit *dynamicUser* auf alle Funktionen der neuen Klasse zugreifen.

```
private final DynamicChoice dynamicUser = new DynamicChoice();
```

3.5.1 Datenverfolgung

Um Diagramme automatisch aufrufen zu lassen, müssen erst mal die Diagrammaufrufe eines Nutzers verfolgt werden. Dazu wird jedes Mal, wenn ein neuer Diagrammtyp manuell ausgewählt wurde, die Funktion *addList(diagrammName)* aufgerufen. Mit dieser Funktion wird der Name des Diagrammtyps zur Speicherung übertragen.

```
//speichern eines Diagrammaufrufs
dynamicUser.addList(diagrammName);
```

In der Methode *addList* werden die Diagrammaufrufe der Nutzer gezählt und in der Hashmap des Nutzers gespeichert. Die Hashmaps der einzelnen Nutzer heißen *clickMap* und werden in einer weiteren Hashmap *userMap* mit dem Nutzernamen als Schlüssel organisiert. In der *clickMap* sind die Namen der Diagrammtypen die Schlüssel und die Anzahl der Aufrufe die veränderlichen Werte. Die Methode geht auch darauf ein, wenn der Nutzer noch nicht in der *userMap* angelegt ist oder der Diagrammtyp in der *clickMap* noch nicht vorhanden ist.

```
public void addList(final String diagrammID) {
    //Diagrammtyp für die gesamte Klasse speichern
    diagramm = diagrammID;
    //Wenn der Nutzer angelegt wurde
    if (userMap.containsKey(user)) {
        //Wenn der Diagrammtyp vorhanden ist, Zähler aufaddieren
        if (userMap.get(user).containsKey(diagrammID)) {...}
        //ansonsten Diagrammtyp hinzufügen
        else {...}
    }
    //andernfalls neue clickMap anlegen
    else {...}
    //Daten speichern
    writeFile();
}
```

Zu Erst wird der übergebene Diagrammtyp in einer globalen Variablen abgespeichert, um ihn für alle Methoden zur Verfügung zu haben. Danach wird überprüft, ob der Nutzer schon in der *userMap* registriert ist. Wenn dies nicht der Fall ist, wird eine neue *clickMap* für den Nutzer angelegt.

```
//neue clickMap anlegen
final HashMap<String, Integer> clickMap =  new HashMap<>();
//clickMap der userMap hinzufügen
userMap.put(user, clickMap);
```

Wenn der Nutzer in der *userMap* schon vorhanden ist, muss als nächstes überprüft werden, ob der übergebene Diagrammtyp schon einmal aufgerufen wurde. Wenn der Diagrammtyp schon einmal aufgerufen wurde, also in der *clickMap* des Nutzers vorhanden ist, wird der Aufrufzählerwert der in der *clickMap* hinterlegt ist um eins erhöht.

```
//alten Aufrufwert abfragen
final int click = (int) userMap.get(user).get(diagrammID);
//Aufrufwert um eins erhöhen und hinzufügen
userMap.get(user).put(diagrammID,  click + 1);
```

Wurde der Diagrammtyp noch nie aufgerufen, wird er der *clickMap* hinzugefügt und mit einem Aufruf initialisiert.

```
userMap.get(user).put(diagrammID, 1);
```

3.5.2 Datennutzung

Die gesammelten Daten werden dazu verwendet, eine personalisierte automatische Diagrammauswahl zu ermöglichen. Dazu werden in der Klasse *DynamicChoice* die beiden Methoden *getUserDiagramm* und *getMostKlicked* bereitgestellt. Die Methode *getUserDiagramm* wird aufgerufen, wenn eine automatische Diagrammauswahl gewünscht wird. In ihr wird zu Erst der String *diagrammID* deklariert und mit der globalen Variablen *diagramm* initialisiert. Anschließend wird überprüft, ob die *clickMap* des Anwenders leer ist. Sind Einträge vorhanden, wird mit der Methode *getMostKlicked* der am häufigsten aufgerufene Diagrammtyp ausgewählt.

```
public String getUserDiagramm() {
    String diagrammID = diagramm;
    if (!userMap.get(user).isEmpty()) {
        //meist aufgerufenen Diagrammtypen auswählen
        diagrammID = getMostClicked(userMap.get(user));
    }
    return diagrammID;
}
```

Es wird direkt die *clickMap* übergeben, damit nicht eine Ebene höher in der Datenstruktur gearbeitet werden muss. Die Methode speichert alle Schlüsselattribute der Hasmap in einem Array ab. Die Schlüssel sind die Diagrammtypen. Dann wird der erste Schlüssel in den String *diagrammName* gespeichert. In einer *for*-Schleife werden danach alle Schlüssel abgefragt, ob deren Aufrufanzahl größer ist, als die des Strings *diagrammName*. Ist dies der Fall wird der String mit dem Schlüsselattribut ersetzt.

```
private String getMostClicked(final HashMap<String, Integer> clickMap) {
    //Schlüsselattribute in Array laden
    final Object[] keys = clickMap.keySet().toArray();
    //Ersten Wert abspeichern
    String diagrammName = (String) keys[0];
    //Schleife über alle Schlüsselattribute
    for (int i = 1; i < keys.length; i++) {
        //Falls ein Wert mehr Aufrufe hat, Wert mit Schlüssel ersetzen
        if (clickMap.get(diagrammName) < clickMap.get(keys[i])) {
            diagrammName = (String) keys[i];
        }
    }
    return diagrammName;
}
```

3.5.3 Nutzerwechsel

Es wird beim EWMS davon ausgegangen, dass der angemeldete Nutzer des Computers auch der Anwender des EWMS ist. Es gibt aber auch die Möglichkeit den Nutzer intern im Diagrammmodul zu wechseln. Der Nutzerwechsel wird mit Hilfe eines Menüknopfs in der Menüleiste des Fensters gestartet. Wenn der Knopf betätigt wird, führt er die Methode *setUser* aus. Diese Methode startet ein Eingabefenster zur Eingabe des neuen Nutzernamens und gibt einen Booleanwert zurück. Es wird nicht der neue Nutzername übergeben, da der Nutzer nur für die Nutzerverfolgung interessant ist und die findet in der Klasse *DynamicChoice* statt. Ist dieser Wert *true* wurde der Nutzerwechsel nicht abgebrochen und das Diagrammmodul wird auf Startzustand zurückgesetzt. Dazu wird die Booleanvariable *firstChoice* auf *true* gesetzt, sodass wieder

eine automatische Diagrammauswahl gestartet werden kann. Des Weiteren wird der gesamte Inhalt aus den Registerkarten entfernt und die automatische Diagrammauswahl gestartet. Ist der Booleanwert von *setUser false*, passiert nichts und der alte Nutzer kann ohne Änderungen weiter arbeiten.

```
//mit setUser neuen Nutzer auswählen
//Überprüfen, ob Nutzerwechsel abgebrochen wurde
if (dynamicUser.setUser()) {
    //Wenn Nutzer gewechselt Diagrammmodul auf Ausgangssituation setzen
    firstChoice = true;
    tab.removeAll();
    createOverview();
}
```

Die Methode *setUser* aus der Klasse *DynamicChoice* startet ein einfaches Eingabefenster, das den Nutzer dazu auffordert einen neuen Nutzernamen einzugeben. Der Nutzer hat auch die Möglichkeit diesen Vorgang abzubrechen.

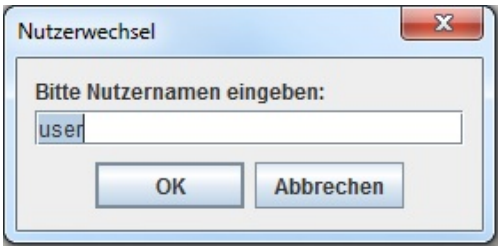

Abbildung 3.8: Eingabefenster zum Nutzerwechsel

```
public boolean setUser() {
    //Eingabefenster starten
    final Object input = JOptionPane.showInputDialog(null,
        "Bitte Nutzernamen eingeben:", "Nutzerwechsel",
        JOptionPane.PLAIN_MESSAGE, null, null, "user");
    if (input != null) {
        user = input.toString();
        checkUserInList();
        return true;
    }
    return false;
}
```

Der Nutzername der Eingegeben wird, wird in der Objektvariable *input* gespeichert. Anschließend prüft eine *if*-Abfrage ob *input* nicht *null* enthält. Wenn der Nutzer sich dazu entschieden hat, den Nutzerwechsel abzubrechen, wird *input* mit *null* gefüllt. Wenn ein neuer Nutzername vorliegt, wird die Objektvariable in ein String umgewandelt und in die globale Variable *user* geschrieben. Danach wird die Funktion *checkUserInLIst* aufgerufen, die überprüft, ob der Nutzer in der *userMap* vorhanden ist. Wenn der Nutzer nicht vorhanden ist, wird die Methode *addList(null)* aufgerufen. In dieser Methode wird die Aufgabe zum hinzufügen neuer Nutzer ausgeführt, da der Nutzer noch nicht in der *userMap* vorhanden ist. Zu Letzt wird in der Abfrage *true* zurückgegeben und damit signalisiert, dass der Nutzerwechsel erfolgreich war. Wurde der Nutzerwechsel abgebrochen, wird direkt *false* zurückgegeben.

3.5.4 Datenspeicherung

Damit die gesammelten Daten nicht verloren gehen, wenn das Programm geschlossen wird, werden sie nach jeder Änderung in eine Datei gespeichert. Möchte man die *userMap* speichern, wird die Methode *writeFile* verwendet. Diese Methode generiert eine Datei die mit der *userMap* gefüllt wird. Der Name der Datei ist mit *Userlist* festgelegt und wird immer am gleichen Ort abgespeichert. Wenn eine Änderung der *userMap* gespeichert werden soll, wird die alte Datei überschrieben.

```
private void writeFile() {
    try {
        //Ausgangsdatei erzeugen
        final FileOutputStream outFile = new FileOutputStream(fileName);
        //Funktion zum Schreiben der Daten in die Datei
        final ObjectOutputStream output = new ObjectOutputStream(outFile);
        //Daten in Datei schreiben
        output.writeObject(userMap);
        output.close();
    }
    catch (final IOException e) {
        //Ausnahmebehandlung
        e.printStackTrace();
    }
}
```

Mit Hilfe der Methode *readFile* werden die gesicherten Daten wieder eingelesen. Die Daten werden nur bei Aufruf der Klasse eingelesen, damit die Daten aus den Hashmaps während der Ausführung des Programms nicht verloren gehen.

Zuerst wird die Datei eingelesen. Der Dateiname ist der Gleiche wie bei der Methode *writeFile*, da er vorab festgelegt wird. Nach dem Einlesen der Datei wird eine Funktion gestartet, die die Datenübertragung zwischen Datei und Programm ermöglicht. Mit der Methode *readObject* werden die Daten in die *userMap* geschrieben. Ein Zusatz vor dem Befehl gibt an, was für Datentypen eingelesen werden.

```
private void readFile() {
    try {
        //Datei einlesen
        final FileInputStream inFile = new FileInputStream(fileName);
        //Funktion zum Lesen der Datei
        final ObjectInputStream input = new ObjectInputStream(inFile);
        //Daten in userMap übernehmen
        userMap = (HashMap<String, HashMap>) input.readObject();
        input.close();
    }
    catch (final ClassNotFoundException | IOException e) {
        //Fehlermeldung bei fehlender Datei
        System.out.println("File not Found.");
    }
}
```

Um die Programmabläufe beider Methoden liegt ein try-catch-Block. Ein try-catch-Block dient zur Fehlerbehandlung von möglicherweise auftretenden Fehlern. Er muss immer um einen Programmabschnitt gelegt werden, wenn man nicht alle möglichen Fehler von vorn herein verhin-

dern kann. Im try-Block werden die eigentlichen Befehle ausprobiert. Wenn bei diesen Befehlen ein Fehler auftritt, stürzt das gesamte Programm nicht ab. Stattdessen wird der catch-Block ausgeführt. “This class is the general class of exceptions produced by failed or interrupted I/O operations.” [2] Im catch-Block wird dann eine Ausnahmebehandlung oder englisch Exception geworfen. Diese Exception behandelt dann den jeweiligen Fehler, in den meisten Fällen wird eine Fehlermeldung ausgegeben. In den beiden Fällen des Einlesens und des Schreibens von Dateien wird eine IOException gebraucht. Passiert ein Fehler beim Einlesen oder Schreiben, wie zum Beispiel das nicht Vorhandensein einer einzulesenden Datei, wird diese Exception ausgeführt.

3.6 Entwicklung einer grafischen Oberfläche

Die grafische Oberfläche des Diagrammmoduls ist schlicht gehalten und besteht aus einer Menüleiste und einem JTabbedPane. Ein JTabbedPane ist ein Panel mit Registerkarten oder Tabs aus der Swing-Bibliothek von Java. Die Swing-Bibliothek, ein Teil der Java Foundation Classes (JFC), beherrscht das Zeichnen grafischer Grundelemente, bietet grafische Komponenten wie Fenster und Schaltflächen und definiert ein Modell zu Behandlung von Ereignissen.[3] In den Registerkarten des Diagrammmoduls werden die Übersichten und die Diagramme dargestellt. Jedes Diagramm bekommt eine eigene Registerkarte, sodass der Nutzer schnell zwischen den Diagrammen wechseln kann, ohne dass diese jedes Mal neu erstellt werden müssen.

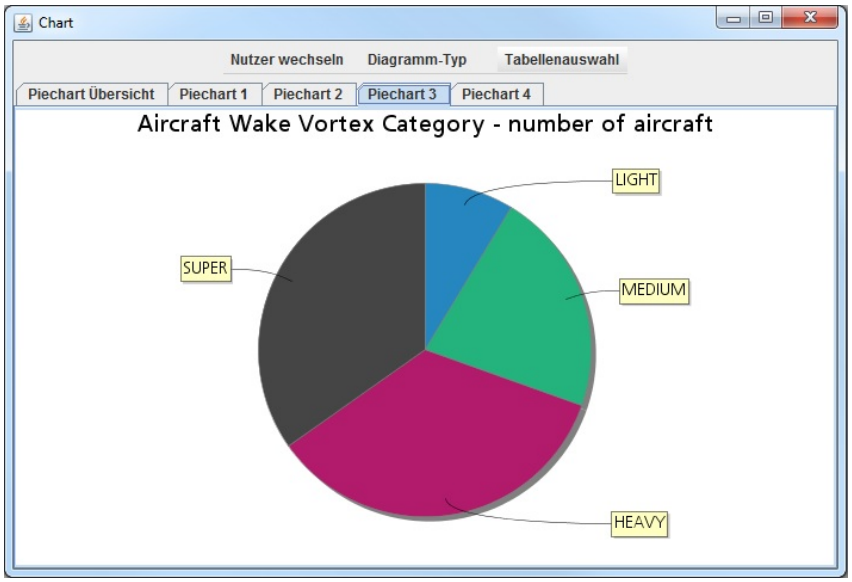

Abbildung 3.9: Beispiel zur grafischen Oberfläche

3.6.1 Das Hauptfenster

Das Hauptfenster, die Menüleiste und das Registerkartenpanel werden global deklariert, zur Verfügbarkeit in der ganzen Klasse. Die Eigenschaften des Hauptfensters werden im Konstruk-

[2] Beschreibung der Klasse IOException,
`http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html`
[3] vgl. Ullenboom, C. (2012): Java ist auch nur eine Insel. 19 Grafische Oberflächen mit Swing

tor der Klasse beschrieben. Es wird festgelegt, wie die Elemente angeordnet und organisiert sind und die Erscheinungsform beim Start des Diagrammmoduls aussieht.

Der Layoutmanager des Hauptfensters ist ein Borderlayout. “Ein Layoutmanager ist dafür verantwortlich, Elemente eines Containers nach einem bestimmten Verfahren anzuordnen, zum Beispiel zentriert oder von links nach rechts.” [4] Das BorderLayout richtet die einzelnen darzustellenden Objekte nach Himmelsrichtungen am Rand oder im Zentrum des Fensters aus. Die Menüleiste wird in einem extra Panel erzeugt, da sie den Standardlayoutmanager zur Anzeige braucht. Dies ist ein Flowlayout, das die Objekte von links nach rechts in ihrer Standardgröße anordnet. Das Menüpanel wird im Norden des Fensters, also oben angeordnet und das JTabbedPane im Zentrum.

```
//Layoutmanager festlegen
frame.setLayout(new BorderLayout());
//menuPanel oben anordnen
frame.add(menuPanel, BorderLayout.NORTH);
//tab im Zentrum anordnen
frame.add(tab, BorderLayout.CENTER);
```

Die Erscheinungsform des Hauptfensters beim Start ist die an den Bildschirm angepasste maximierte Darstellung. Außerdem wurde noch die Standardgröße des Fensters festgelegt und die Eigenschaft, dass das Programm beendet wird, wenn man auf den Schließenbutton klickt.

```
//Standardgröße festlegen
frame.setSize(1450, 960);
//Startgröße an Bildschirm anpassen
frame.setExtendedState(JFrame.MAXIMIZED_BOTH);
//Beenden des Programms beim Schließen
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

3.6.2 Die Registerkarten

Die Organisation der Diagramme in Registerkarten erfolgt mit einem JTabbedPane. Die Diagramme und die Übersichten bekommen alle eine eigene Registerkarte. Mit Hilfe des Befehls *tab.addTab("Titel", Inhalt);* kann Inhalt dem JTabbedPane hinzugefügt werden. Wenn die Diagrammübersicht dargestellt wird, werden alle Tabs vorher entfernt und die Übersicht in einem einzelnen Tab gestartet.

Abbildung 3.10: Die Registerkarten

Nachdem der Nutzer dann einen Diagrammtypen ausgewählt hat, wird die Methode *addOrUpdateChartTab* ausgeführt. Dies ist auch eine überladene Methode. Es gibt hier eine Version, bei der ein Diagramm mitgegeben wird und eine, bei der nichts übergeben wird. Der Klick auf einen Diagrammtypen öffnet die Version ohne Übergabediagramm und daraufhin öffnet sich die Diagrammübersicht über Tabellen. Dazu wird die Methode *addChartTab* aufgerufen und ihr ein Titel und die Übersicht in einem Panel mit Scrollbalken übergeben. Gibt es nur eine Eingangstabelle, wird gleich das dazugehörige Diagramm geladen.

[4] Ullenboom, C. (2012): Java ist auch nur eine Insel. 19.11 Alles Auslegungssache: Die Layoutmanager

```
private void addChartTab(final String title, final JPanel chartPanel) {
    //Index vom Titel abfragen
    final int index = tab.indexOfTab(title);
    //Grö ß e vom Index prüfen
    if (index < 0) {
        //Wenn kleiner 0, neuen Tab erstellen
        tab.addTab(title, chartPanel);
    }
    else {
        //andernfalls Tab ersetzen
        tab.setComponentAt(index, chartPanel);
    }
    //neuen Tab anzeigen
    tab.setSelectedIndex(tab.indexOfTab(title));
}
```

Wenn direkt ein Diagramm geöffnet wird, legt die zweite Version der Methode *addOrUpdateChartTab* den Titel des neuen Tabs fest und führt die Funktion *addChartTab* aus. Diese Funktion überprüft, ob ein Tab schon vorhanden ist, indem der Index des Titels unter den Registerkarten rausgesucht wird. Ist die Registerkarte noch nicht vorhanden, wird der Index auf -1 gesetzt. Eine *if*-Abfrage prüft nun, ob der Index kleiner Null ist. Wenn der Index kleiner Null ist, wird ein neuer Tab hinzugefügt, andernfalls wird die alte Registerkarte mit einem neuen Diagrammpanel ersetzt. Dies dient dazu, die angezeigten Daten immer aktuell zu halten. Am Schluss der Funktion wird noch die neu erstellte Registerkarte als angezeigte Registerkarte ausgewählt.

Dem JTabbedPane ist auch noch ein ActionListener angehängt und zwar ein ChangeListener, der auf Änderungen reagiert. Jedes Mal, wenn ein Tab gewechselt wird, wird die Tabellennummer des dargestellten Diagramms abgefragt. Dazu wird überprüft, ob der Index des angezeigten Tabs größer Null ist. Damit der Index größer Null ist, muss ein Tab ausgewählt sein, dies ist unter fehlerfreien Umständen immer der Fall. Dann wird der Titel der Registerkarte abgefragt und das letzte Zeichen davon genommen. Wenn dieses Zeichen eine Zahl ist, entspricht es der um eins erhöhten Tabellennummer. Also wird das Zeichen dann in einen Integer umgewandelt und um Eins verringert als Tabellennummer abgespeichert.

```
//Auszuführender Code bei Änderungen der Tabs
if (tab.getSelectedIndex() >= 0) {
    //Titel abfragen
    final String title = tab.getTitleAt(tab.getSelectedIndex());
    //letztes Zeichen nehmen
    final String lastChar = title.substring(title.length() - 1);
    //Auf Zahl prüfen
    if (lastChar.matches("\\d*")) {
        //Wenn Zahl, dann als Tabellennummer speichern
        tableNumber = Integer.parseInt(lastChar) - 1;
    }
}
```

3.6.3 Die Menüleiste

In der Menüleiste finden sich ein Menü zu Auswahl der Eingangstabellen und zwei Menüknöpfe. Einer der beiden Menüknöpfe öffnet die Diagrammauswahl, mit dem anderen wechselt man den Nutzer. Die Menüleiste und die Menüobjekte sind wieder Swing-Komponenten.

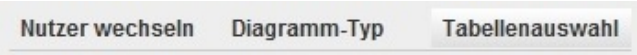

Abbildung 3.11: Die Menüleiste

In der Methode *createMenuBar* wird die Methode erstellt. Diese Methode findet jedes Mal Anwendung, wenn ein Diagrammtyp ausgewählt wurde. Deshalb ist der erste Schritt in der Methode das Löschen des Inhaltes der Menüleiste. Danach werden die beiden Knöpfe erstellt und mit ActionListenern ausgerüstet, die die jeweilige Funktion ausführen.

Dann kommt das Menü, in dem die Tabellenauswahl stattfindet. Im ersten Schritt wird eine JMenu-Variable deklariert und die Anzahl der Diagramme, die für einen Tabellentypen vorliegen, ausgelesen.

```
final JMenu diagramm = new JMenu("Tabellenauswahl");
final int noOfTables = dataSetMap.get(diagrammName).size();
```

Im nächsten Schritt werden die Menüpunkte dynamisch in einer *for*-Schleife erstellt. Die Anzahl der Durchläufe entspricht der Anzahl vorliegender Diagramme.

```
for (int i = 0; i < noOfTables; i++) {
    //Menüitem erstellen mit dynamischem Namen
    final JMenuItem dataChoice = new JMenuItem("Tabelle " + (i + 1));
    final int y = i;
    //ActionListener erstellen
    dataChoice.addActionListener(new ActionListener() {...});
    //Wenn mehrere Tabellen als Grundlage, Name ändern
    if (checkStacked && i == (noOfTables - 1)
            && (diagrammName.startsWith("Line")
            || diagrammName.startsWith("XY"))) {
        dataChoice.setText(diagrammName + " Zusammenfassung");
    }
    //Menüitem dem Menü hinzufügen
    diagramm.add(dataChoice);
}
```

In der Schleife werden die Menüpunkte erstellt und mit dynamischem Namen versehen. Der Name beinhaltet die um Eins erhöhte Nummer der Tabelle, solange nur eine Tabelle als Grundlage des Diagramms vorlag. Dies wird an einem Booleanwert und dem Namen des Diagrammtypen erkannt. Wenn bei einem Diagrammtyp die Möglichkeit besteht, dass mehrere Tabellendaten ein Diagramm erzeugten, wird der Boolean auf *true* gesetzt. Außerdem wird den Items noch jeweils ein ActionListener mitgegeben, der die Methode *addOrUpdateChartTab* mit einer Diagrammübergabe ausführt.

Als letzter Schritt wird ein Menüitem hinzugefügt, der die Diagrammübersicht über Tabellen aufruft. Dieser Menüknopf wird nur hinzugefügt, wenn mehr als eine Eingangstabelle vorliegt.

```
if (noOfTables > 1) {
    //Wenn mehr als eine Tabelle, Menuitem erstellen
    final JMenuItem tableOverview = new JMenuItem(diagrammName + " Übersicht");
    //ActionListener hinzufügen
    tableOverview.addActionListener(new ActionListener() {...});
    //JMenuItem dem JMenu hinzufügen
    diagramm.add(tableOverview);
}
```

Kapitel 4

Beschreibung der Ergebnisse

Das Diagrammmodul ist ein erweiterbares Modul zur Diagrammauswahl und -anzeige des neuen EWMS 2. Die Tests wurden mit Hilfe von implementierten Testdaten durchgeführt, da das EWMS 2 im Entwicklungszeitraum des Diagrammmoduls noch keine Daten verarbeiten kann.

Was die Testdaten sind und was das Diagrammmodul machen kann wird im Folgenden beschrieben.

4.1 Vorstellung der Testdaten

Die Testdaten sind in eigenen Klassen fest implementiert. Diese Testklassen geben die Daten so weiter, wie es ein Auswertungsmodul machen würde. Sie werden als eigener Task gestartet und vor das Diagrammmodul geschaltet, sodass die Ausgangsdaten der Testklassen die Eingangsdaten des Diagrammmoduls sind. Sie beinhalten die Eingangstabellen in unterschiedlicher Größe mit Tabellenbeschreibungen. Um möglichst viele Fälle abzudecken gibt es vier Testklassen mit unterschiedlicher Anzahl an Tabellen und Tabellenspalten, verschiedenen Tabellenstrukturen und differierenden Datenmengen pro Tabelle. Nur die Anzahl an Spalten ist für die Diagrammerstellung interessant, da sie verschiedene Datentypen enthalten. Die Anzahl der Zeilen spiegeln die Menge der Daten wieder. Die Tabellen sind in Arrays gespeichert.

Eine Tabelle wird in drei Schritten erzeugt. Zuerst wird die Tabellendefinition festgelegt, in der bestimmt wird von welchem Datentyp welche Spalte ist. Danach wird ein zweidimensionales Array erstellt, in dem die Daten gespeichert werden.

```
//Definition der Datentypen in den Spalten
final DataType[] definition = { DataType.AIRCRAFT_WVC,
    DataType.AIRCRAFT_COUNT };
```

Danach wird ein zweidimensionales Array erstellt, in dem die Daten gespeichert werden.

```
//Inhalt der Tabellen in einem Array
final Object[][] dataArray = new Object[][] {
    { WakeVortexCategory.LIGHT, 2 },
    { WakeVortexCategory.MEDIUM, 513 },
    { WakeVortexCategory.HEAVY, 127 },
    { WakeVortexCategory.SUPER, 8 }, };
```

Abschließend werden die Daten und die Definition in einem Datencontainer zusammengepackt und mit dem Befehl *addOutput* zu den Ausgabecontainern hinzugefügt.

```
//Tabelle hinzufügen zu Ausgabe
addOutput(new DataContainer(new DataDefinition(definition), dataArray));
```

Alle Tabellen werden mit diesen drei Schritten erzeugt.

4.1.1 Testszenario Wirbelschleppen

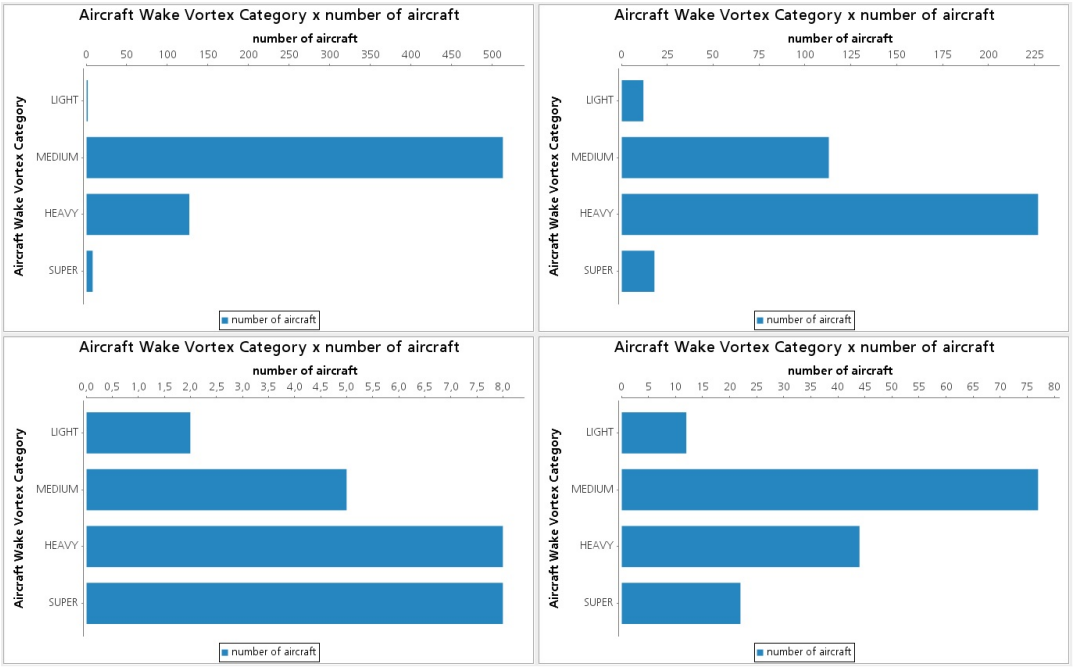

Abbildung 4.1: Szenario mit mehreren Tabellen

Das erste Testszenario gibt vier Tabellen mit je zwei Spalten aus. Eine Spalte enthält Zahlenwerte. Es beschreibt die Menge an Flugzeugen bestimmter Wirbelschleppenkategorien die eine Simulation durchlaufen haben. Wirbelschleppen sind Luftverwirbelungen die Flugzeuge im Flug erzeugen. “Als Wirbelschleppen werden die beiden an den Flügelspitzen und Klappen entstehenden, gegenläufigen Luftwirbel bezeichnet, die jedes Flugzeug auf seiner Flugbahn hinterlässt.”[1] Je größer das Flugzeug ist, desto größer sind die Wirbelschleppen. Deshalb sortiert man Flugzeuge in unterschiedliche Wirbelschleppenkategorien ein. Dieses Szenario testet vor allen Dingen die Bearbeitungsmöglichkeiten mehrerer Tabellen gleichzeitig. Wichtig in diesem Szenario ist die Menge an Tabellen, die verarbeitet werden sollen. Alle Eingangstabellen werden vom Diagrammmodul beachtet und in einer eigenen Diagrammübersicht über Tabellen gemeinsam dargestellt.

4.1.2 Testszenario Dauer Flugphasen

Das zweite Szenario gibt eine Tabelle mit 3 Spalten wieder. Hier enthält ebenfalls eine Spalte Zahlenwerte und die anderen beiden Kategorien. Es wird die durchschnittliche Dauer beschrieben, die Flugzeuge in einem bestimmten Flugtyp und Flugphase benötigen. Diese Testklasse testet auf die Beachtung mehrerer Kategorien bei der Diagrammerzeugung. Diese werden in einem geschachtelten Balkendiagramm ausgegeben, in der die Kategorien unterschiedliche Balkenfarben haben, oder in Kuchendiagrammen, bei denen die Kategorien einer Reihe zusammengefasst werden zu einer Superkategorie.

[1] Fraport: FAQ Wirbelschleppen. Wie entstehen Wirbelschleppen?, `http://www.fraport.de/content/fraport/de/nachhaltigkeit/schallschutz-fluglaerm/wirbelschleppen/faq-wirbelschleppen.html`

4.1.3 Testszenario Gleitender Zeitablauf

Im nächsten Szenario werden in 2 Tabellen mit 3 Spalten gleitende Zeitverläufe dargestellt. Es gibt nun 2 Spalten mit Zahlenwerten, davon eine mit Zeitwerten. Hier wird die Anzahl von Flugbewegungen in ineinander übergehenden Zeitabschnitten beschrieben. Dies wird beachtet, indem Spalten vom Typ Zeit auf Stunden gerundet und in einem Diagramm mit Zeitstrahl gezeichnet werden.

4.1.4 Testszenario Seperationen

Die letzte Testklasse beinhaltet eine Tabelle mit 5 Zahlenwertspalten. Die erste Spalte ist eine Zeitspalte. Das Szenario beschreibt die Häufigkeit verschiedener Flugereignissen, die zu bestimmten Zeiten aufeinander folgen.

```
//Eine Zeitspalte und vier Spalten mit Häufigkeiten zu Flugereignissen
final DataType[] definition = {DataType.TIME,
    DataType.ARRIVAL_ARRIVAL_FREQUENCY,
    DataType.ARRIVAL_DEPARTURE_FREQUENCY,
    DataType.DEPARTURE_ARRIVAL_FREQUENCY,
    DataType.DEPARTURE_DEPARTURE_FREQUENCY};
```

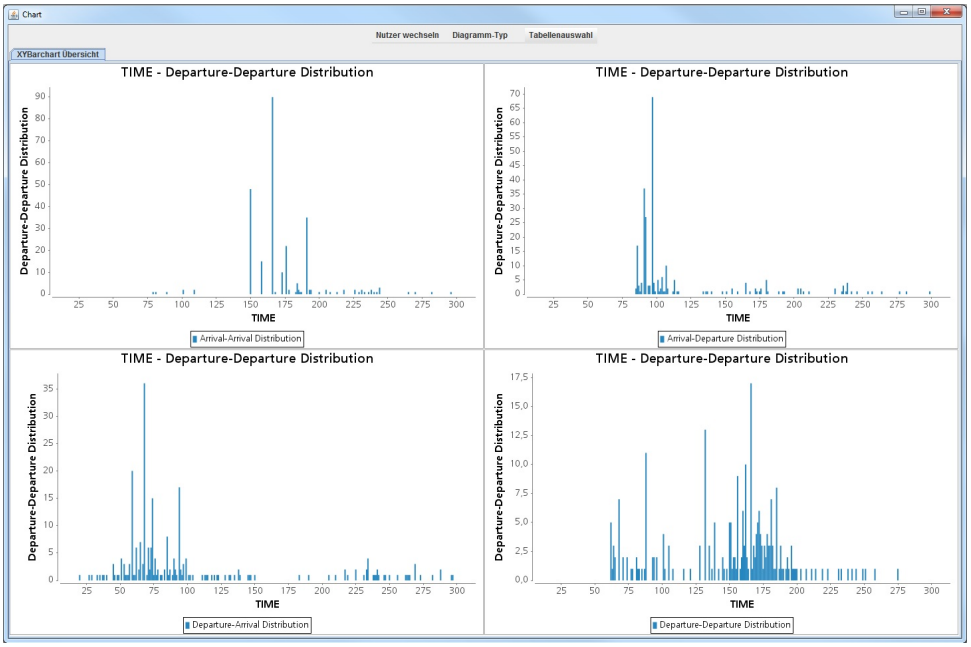

Abbildung 4.2: Szenario mit Zeitverläufen

In diesem Fall ist die Beachtung von vielen Wertspalten gefordert. Solche Tabellen werden so behandelt, als sei eine Wertspalte zusammen mit der ersten Spalte eine Tabelle. Wenn eine Spalte in diesem Fall Zeiten enthält, wird sie nicht als Wertspalte behandelt sondern als Kategoriespalte.

4.2 Was macht das Diagrammmodul?

Das Diagrammmodul baut aus den einheitlich gestalteten Eingangsdaten des EWMS dynamisch mögliche Diagramme auf. Die Eingangsdaten werden mit Hilfe des JFreeChart-Frameworks in geeignete Datenstrukturen umgewandelt.

Der vom Nutzer favorisierte Diagrammtyp wird als erstes verwendet, wenn er verfügbar ist, ansonsten kann er sich einen anderen Diagrammtyp in einer Übersicht aussuchen. Die Übersicht zeigt alle verfügbaren Diagrammtypen an. Es werden keine Beispieldiagramme genutzt, sondern die aus den eingegangen Tabellen erzeugten Diagramme. Dadurch kann der Nutzer einen für seine Daten sinnvollen und favorisierten Diagrammtyp auswählen.

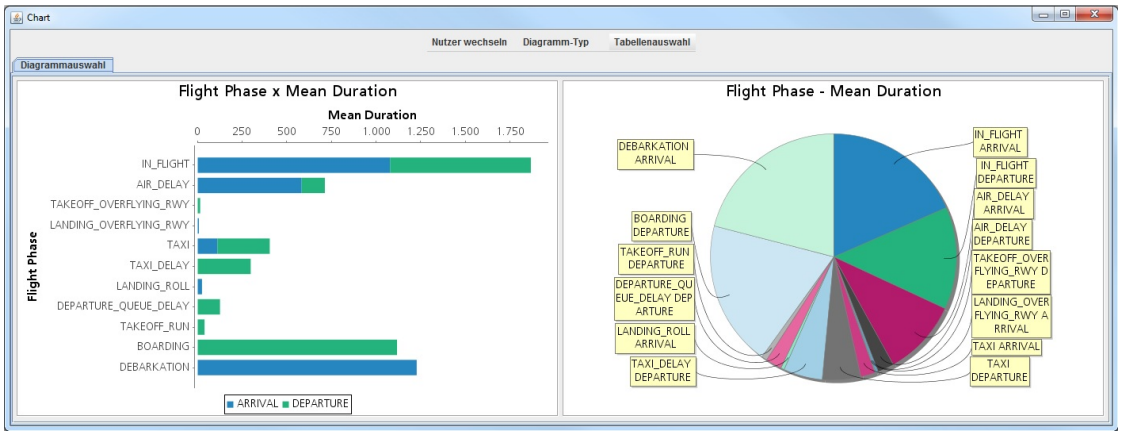

Abbildung 4.3: Diagrammmodul nach dem Start

Nach der Typauswahl werden die Diagramme der verschiedenen Tabellen in einer weiteren Übersicht angezeigt, sodass der Nutzer die Daten der Tabellen visuell vergleichen kann. Jedes Diagramm kann einzeln angezeigt und abgespeichert werden. Für das Modul ist eine einfache grafische Oberfläche entwickelt worden, in der Anwender sich schnell zurecht findet. Es wird das aus den Internetbrowsern bekannte Prinzip der Registerkarten verwendet. Alle neu aufgerufenen Diagramme werden in einer neuen Registerkarte angezeigt.

Damit ein Diagrammtyp automatisch vorab ausgewählt werden kann, werden die Nutzerdaten des Anwenders gesammelt und lokal gespeichert. Für den Fall, das mehrere Nutzer das Programm verwenden, oder die Datenspeicherung auf einen Server ausgelagert wird, ist die Datensammlung personenabhängig. Für diesen Fall ist es auch möglich den Nutzer zu wechseln.

Kapitel 5

Diskussion

In diesem Abschnitt des Berichts werden Ergebnisse der Arbeit diskutiert und beschrieben, welche Zielvorgaben erreicht wurden und welche Arbeitsschritte Probleme hervorbrachten. Die Zielvorgabe dieser Arbeit ist die Entwicklung einer personalisierbaren dynamischen Diagramm-Auswahl für das Extensible Workflow Management for Simulations (EWMS). Zu allen Aufgabenpunkten der Aufgabenstellung wurde eine Lösung erarbeitet.

Das Diagrammauswahlmodul nutzt die existierenden Schnittstellen des EWMS 2, die dieses zur Datenweitergabe bereitstellt. Das Verwenden dieser vorhandenen Programmstrukturen konnte ohne Schwierigkeiten umgesetzt werden. Nicht verwendet werden konnte die Oberfläche des EWMS 2, da dessen Entwicklungsstand zum Zeitpunkt dieser Arbeit noch keine Oberfläche bot. Jedoch wurde eine eigene grafische Oberfläche entwickelt, die sich an den Ideen für das EWMS 2 orientiert.

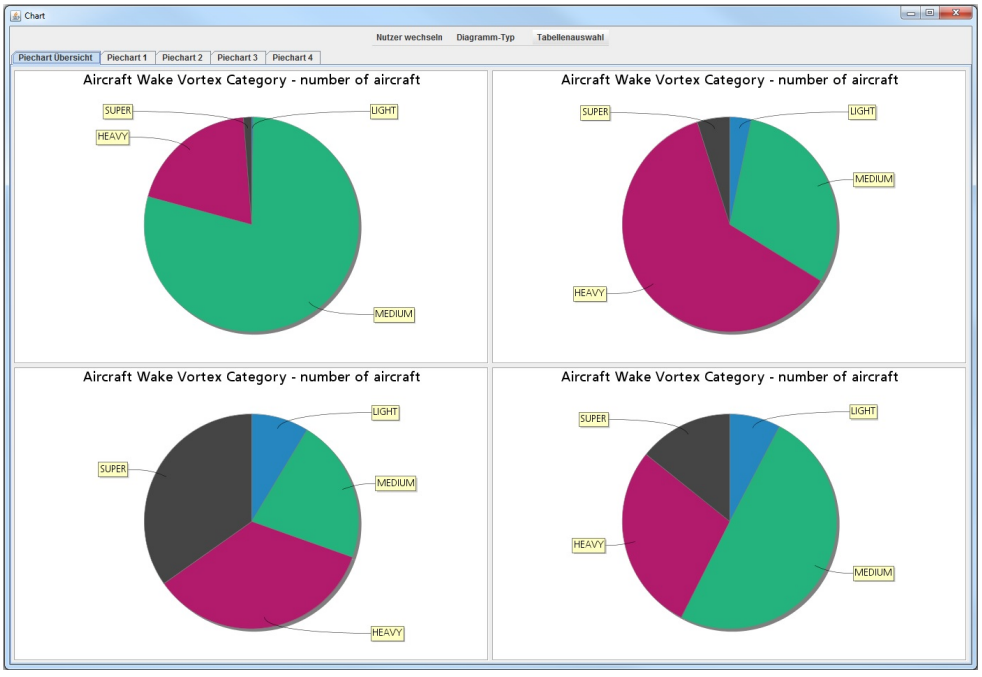

Abbildung 5.1: Mehrere Diagramme im Vergleich

Das Modul verarbeitet die Eingangsdaten zu geeigneten Diagrammen und wählt dynamisch passende Diagrammtypen aus. Es bietet bisher wenige Diagrammtypen, kann aber durch wenig Programmieraufwand erweitert werden. Da für die Tests nur wenige Daten zur Verfügung standen, konnten nicht alle Eventualitäten bei der Implementierung berücksichtigt werden. Alle

Schwierigkeiten, die es mit den Testdaten gibt, werden aber von dem Diagramm-Modul beachtet und gelöst.

Das JFreeChart-Framework wird in großem Umfang und von allen Bereichen der Erstellung der Diagramme verwendet. Vorteilhaft für die dynamische Auswahl geeigneter Diagramme ist, das die meisten Diagrammtypen verschieden strukturierte Datasets benötigen. Auch die Erzeugung mehrerer Diagramme mit JFreeChart funktioniert einwandfrei, da man die fertigen Diagramme in Hashmaps und Listen zwischenspeichern kann.

Das Nutzerverhalten wird beim Verwenden des Diagrammmoduls verfolgt und die Aktivitäten lokal gespeichert. Bisher werden diese nur einmal verwendet, nur wenn das Diagramm-Modul einen Diagrammtyp automatisch auswählen soll. Im Diagrammauswahl-Modul haben sich bisher keine weiteren Anwendungsmöglichkeiten gezeigt. Im Sammeln dieser Daten steckt aber Potenzial für die Zukunft des Moduls, auch durch das Verfolgen der Nutzerdaten im gesamten Auswertungsablauf. Besser gemacht werden kann bei der automatischen Auswahl eines Diagrammtyps das Finden des nächst favorisierten Diagrammtyps, falls der Erste nicht geeignet ist. Zurzeit wird in diesem Fall zur manuellen Übersicht gewechselt.

Die Auswahl sinnvoller Diagrammtypen kann der Anwender am besten entscheiden und deshalb hat er auch die Möglichkeit, zwischen den geeigneten Diagrammtypen in einer Übersicht den für seinen Fall sinnvollsten auszuwählen. Aus dieser Übersicht hat sich auch die Diagrammübersicht über die verschiedenen Eingangstabellen entwickelt. Dadurch kann der Anwender die unterschiedlichen Eingangsdaten direkt vergleichen.

Kapitel 6

Fazit und Ausblick

In den Praxisphasen drei und vier ist mit der Diagrammauswahl ein wichtiger Bestandteil des EWMS 2 entstanden. Es baut mit dem JFreeChart-Framework Diagramme aus Ausgangsdaten von Auswertungsmodulen auf und lässt den Nutzer zwischen geeigneten Diagrammtypen zur Anzeige auswählen. Nun muss nicht mehr für jede Auswertung ein spezifisches Diagramm implementiert werden. Stattdessen ist der Entwickler einer Auswertung in der Lage das Diagrammmodul als Baustein an seine Auswertung anzuhängen.

Von der ersten Generation zur zweiten Generation des EWMS findet ein Philosophiewechsel statt. Statt einem sequentiell ablaufenden Programm mit fest programmierten Elementen wird es ein modular aufgebautes Programm geben, das dem Nutzer ein Baukastenprinzip für die Auswertungserstellung bietet. Das Diagrammmodul wird ein wichtiger und der letzte Baustein einer aufgebauten Auswertung sein.

Für die Zukunft gibt es eine Menge Erweiterungspotenzial in diesem Modul und ich hoffe, das dieses genutzt wird um bestmögliche Auswertungen zu erzeugen.

Für meine nächste Praxisphase freue ich mich auf ein wissenschaftlicheres Thema unter Verwendung und Weiterentwicklung des EWMS.

Abkürzungsverzeichnis

DLR Deutsches Zentrum für Luft- und Raumfahrt e.V.

EWMS Extensible Workflow Management for Simulations

FL Institut für Flugführung

LVS Luftverkehrssysteme

IDE Integrated Development Environment

JFC Java Foundation Classes

Abbildungsverzeichnis

Literaturverzeichnis

[5] Scharnweber, A. , Schier, S. (2010): Das „Extensible Workflow Management for Simulations“ im Einsatz. Braunschweig

[7] Gilbert, D. (2009): The JFreeChart ClassLibrary. Version 1.0.13. Developer Guide.

[8] Ullenboom, C. (2012): Java ist auch nur eine Insel. Bonn: Galileo Press GmbH, 10. Auflage

Quellenverzeichnis

[1] Knabe, F.: Luftverkehrssysteme, `http://www.dlr.de/fl/desktopdefault.aspx/tabid-1136/` Stand: 28.08.2013

[2] o.A.: Was ist die Java-Technologie und wozu wird sie benötigt?, `http://www.java.com/de/download/faq/whatis_java.xml` Stand: 28.08.2013

[3] o.A. (2006): Platform Plug-in Developer Guide, `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fint_eclipse.htm` Stand: 28.08.2013

[4] o.A. (2012): Checkstyle 5.6, `http://checkstyle.sourceforge.net/` Stand: 28.08.2013

[6] Fowler, M. : What is Refactoring?, `www.refactoring.com` Stand: 28.08.2013

[9] o.A. (2013): Class IOException, `http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html` Stand: 28.08.2013

[12] Fraport: FAQ Wirbelschleppen. Wie entstehen Wirbelschleppen?, `http://www.fraport.de/content/fraport/de/nachhaltigkeit/schallschutz-fluglaerm/wirbelschleppen/faq-wirbelschleppen.html` Stand: 28.08.2013